Physical ordered list ~ array

- Memory /address/
  - Garbage collection

- Files (character/byte list/lines in text file,...)

- Disk
  - Disk fragmentation

Lists: Array

\[
L = \text{int}[\text{MAX\_SIZE}]
L[2]=7
\]

2D array

\[
&A[i,j] = A + i*(\text{nr\_el\_in\_row} * \text{el\_size}) + j * \text{el\_size}
\]
Multiple lists, 2-D-arrays, etc...

1 2 3
4 5 6
7 8 9

Linear Lists

- Operations which one may want to perform on a linear list of $n$ elements include:
  - gain access to the $k$th element of the list to examine and/or change the contents
  - insert a new element before or after the $k$th element
  - delete the $k$th element of the list


Abstract Data Type (ADT)

- High-level definition of data types
- An ADT specifies
  - A collection of data
  - A set of operations on the data or subsets of the data
- ADT does not specify how the operations should be implemented
- Examples
  - vector, list, stack, queue, deque, priority queue, table (map), associative array, set, graph, digraph

ADT

- A datatype is a set of values and an associated set of operations
- A datatype is abstract iff it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume

Abstract data types:

- Dictionary (key,value)
- Stack (LIFO)
- Queue (FIFO)
- Queue (double-ended)
- Priority queue (fetch highest-priority object)
- ...

Dictionary

- Container of key-element (k,e) pairs
- Required operations:
  - insert( k,e ),
  - remove( k ),
  - find( k ),
  - isEmpty()
- May also support (when an order is provided):
  - closestKeyBefore( k ),
  - closestElemAfter( k )
- Note: No duplicate keys
Abstract data types

- Container
- Stack
- Queue
- Deque
- Priority queue

- Dictionary/Map/Associative array
- Multimap
- Set
- Multiset
- String
- Tree
- Graph
- Hash

Some data structures for Dictionary ADT

- Unordered
  - Array
  - Sequence/list
- Ordered
  - Array
  - Sequence (Skip Lists)
  - Binary Search Tree (BST)
  - AVL trees, red-black trees
  - (2,4) Trees
  - B-Trees
- Valued
  - Hash Tables
  - Extendible Hashing

Primitive & composite types

Primitive types
- Boolean (for boolean values True/False)
- Char (for character values)
- int (for integral or fixed-precision values)
- float (for storing real number values)
- Double (a larger size of type float)
- String (for string of chars)
- Enumerated type

Composite types
- Array
- Record (also called tuple or struct)
- Union
- Tagged union (also called a variant, variant record, discriminated union, or disjoint union)
- Plain old data structure

Linear data structures

Arrays
- Array
- Bidirectional map
- Bit array
- Bit field
- Bitboard
- Bitmap
- Circular buffer
- Control table
- Image
- Dynamic array

Lists
- Doubly linked list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VList
- XOR linked list
- Zipper
- Doubly connected edge list

Trees

- Binary tree
  - ADT
  - AVL
  - Balanced
  - Binary
  - Decision tree

- Fahrenheit
  - Fenwick tree
  - Exponential
  - Enfilade

- Fusion tree
- Disjoint-set data structure
  - Spaghetti stack
  - SPQR-tree
  - Link/cut tree
  - Ternary search tree
  - Ctrie
  - X-fast

- Boas trees
  - Space-saving
  - Data structure

- Prefix hash tree
  - Hash tree
  - Hash table
  - Hash array
  - Hash list
  - Hash table
  - Hash tree
  - Koenie
  - Prefix hash tree

Hashes, Graphs, Other

- Hashes
  - Bloom filter
  - Distributed hash table
  - Hash array mapped
  - Trie
  - Hash list
  - Hash table
  - Hash tree
  - Koordie
  - Prefix hash tree

- Graphs
  - Adjacency list
  - Adjacency matrix
  - Graph-structured stack
  - Scene graph
  - Binary decision diagram
  - Zero suppressed decision diagram
  - And-inverter graph
  - Directed graph

- Directed acyclic graph
- Propositional directed acyclic graph
- Multigraph
- Hypergraph

- Other
  - Lightmap
  - Winged edge
  - Quad-edge
  - Routing table
  - Symbol table
Lists: Array

```
size
+---------------------+
  | 0 1 3 6 7 5 2     |
  | 0 1 3 6 7 8 5 2     |
  +---------------------+
```

*array (memory address)

- Access i: $O(1)$
- Insert to end: $O(1)$
- Delete from end: $O(1)$
- Insert: $O(n)$
- Delete: $O(n)$
- Search: $O(n)$

Linear Lists

- Other operations on a linear list may include:
  - determine the number of elements
  - search the list
  - sort a list
  - combine two or more linear lists
  - split a linear list into two or more lists
  - make a copy of a list

Stack

- push(x) -- add to end (add to top)
- pop() -- fetch from end (top)

- $O(1)$ in all reasonable cases 😊
- LIFO – Last In, First Out

Linked lists

- Singly linked
- Doubly linked

Linked lists: add

- Adds a new element to the end of the list.
**Linked lists: delete (+ garbage collection?)**

- Doubly-linked list:
  - `head` and `tail`
  - `size`

- Singly-linked list:
  - `head` or `tail`
  - `size`

**Improving Run-Time Efficiency**

- We can improve the run-time efficiency of a linked list by using a doubly-linked list:

  Singly-linked list:
  - `Line_head` → node1 → node2 → ... → nodek → `Line_tail`

  Doubly-linked list:
  - `Line_head` → node1 → ← node2 → ... ← nodek → `Line_tail`

  - Improvements at operations requiring access to the previous node
  - Increases memory requirements...

**Operations**

- Array indexed from 0 to \( n - 1 \):
  
<table>
<thead>
<tr>
<th>( k )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insert before or after the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

- Singly-linked list with head and tail pointers:
  
<table>
<thead>
<tr>
<th>( k )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insert before or after the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

  *under the assumption we have a pointer to the \( k \)th node, \( O(n) \) otherwise

**Improving Efficiency**

- Singly-linked list:
  
<table>
<thead>
<tr>
<th>( k )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insert before or after the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

- Doubly-linked list:
  
<table>
<thead>
<tr>
<th>( k )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insert before or after the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

*under the assumption we have a pointer to the \( k \)th node, \( O(n) \) otherwise

**Introduction to linked lists**

- Consider the following struct definition:

  ```c
  struct node {
    string word;
    int num;
    node *next; //pointer for the next node
  };
  node *p = new node;
  ```
Introduction to linked lists: **inserting a node**

- `node *p;`
- `p = new node;`
- `p->num = 5;`
- `p->word = "Ali";`
- `p->next = NULL`

```
5 Ali num word next
```

Introduction to linked lists: **adding a new node**

- How can you add another node that is pointed by `p->next`?

```
node *p; p = new node; p->num = 5; p->word = "Ali"; p->next = NULL;
node *q; q = new node; q->num = 8; q->word = "Veli";
p->next = q; q->next = NULL;
```

```
5 Ali num word next
7 7 num word link
```

```
5 Ali num word link
8 Veli num word link
```

Introduction to linked lists

```
p = new node; p->num = 5; p->word = "Ali"; p->next = NULL;
q = new node; q->num = 8; q->word = "Veli";
p->next = q; q->next = NULL;
```

```
5 Ali num word link
8 Veli num word link
```

Introduction to linked lists

```
p = new node; delete p;
p = new node[20];
p = malloc(sizeof(node) ); free p;
```

```
p = malloc( sizeof( node ) * 20 );
(p+10)->next = NULL; /* 11th elements */
```
Book-keeping

• **malloc, new** – “remember” what has been created `free(p), delete` (C/C++)
• When you need many small areas to be allocated, reserve a big chunk (array) and maintain your own set of free objects
• Elements of *array of objects* can be pointed by the pointer to an object.

Object

• Object = new object_type
• Equals to creating a new object with necessary size of allocated memory (delete can free it)

---

Some links

• **Pointer basics:**
• C++ Memory Management : What is the difference between malloc/free and new/delete?

Alternative: **arrays and integers**

• If you want to test pointers and linked list etc. data structures, but do not have pointers familiar (yet)
• Use arrays and indexes to array elements instead...

---

Replacing pointers with array index

```
head=3
next   key   prev
7  5  1
5  8  4
```

```
new = free;
free = next[free];
free object x
next[x]= free
free = x
```

Maintaining list of free objects

```
head=3
next   key   prev
7  5  1
5  8  4
```

```
free = -1 => array is full
allocate object:
new = free;
free = next[free];
free object x
next[x]= free
free = x
```
Multiple lists, single free list

head1=3
head2=6
free =>2

next
key
prev

Hack: allocate more arrays ...

use integer division and mod
AA((i-1)/7) => [ (i-1) % 7 ]
LIST(10) = AA[ 1 ][ 2 ]
LIST(19) = AA[ 2 ][ 5 ]

Queue (FIFO)

Queue (basic idea, does not contain all controls!)
First = List[F]
Last = List[L-1]
Full: return (L == MAX_SIZE)
Empty: F < 0 or F >= L

Circular buffer

A circular buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.
Circular Queue

L F MAX_SIZE-1
1 6 7 5 2

First = List[F]
Add_to_end( x ) : { List(L=x; L= (L+1) % MAX_SIZE ) } % L = L % MOD
Last = List( (L-1+MAX_SIZE) % MAX_SIZE )
Full: return ( (L+1)%MAX_SIZE == F )
Empty: F==L

Queue

• enqueue(x) - add to end
• dequeue() - fetch from beginning

FIFO – First In First Out

• O(1) in all reasonable cases 😊

Stack

• push(x) -- add to end (add to top)
• pop() -- fetch from end (top)

• O(1) in all reasonable cases 😊
• LIFO – Last In, First Out

Stack based languages

• Implement a postfix calculator – Reverse Polish notation
  5 4 3 * 2 + ➞ 5+(4*3)-2

• Very simple to parse and interpret
• FORTH, Postscript are stack-based languages

Array based stack

• How to know how big a stack shall ever be?

3 6 7 5
3 6 7 5 2

• When full, allocate bigger table dynamically, and copy all previous values there
• O(n) ?

• When full, create 2x bigger table, copy previous n elements:

• After every 2^k insertions, perform O(n) copy
• O(n) individual insertions +
• n/2 + n/4 + n/8 ... copy-ing
• Total: O(n) effort!
What about deletions?

- when \( n = 32 \rightarrow 33 \) (copy 32, insert 1)
- delete: 33 ->32
  - should you delete immediately?
  - Delete only when becomes less than 1/4th full

- Have to delete at least \( n/2 \) to decrease
- Have to add at least \( n \) to increase size
- Most operations, \( O(1) \) effort
- But few operations take \( O(n) \) to copy
- For any \( m \) operations, \( O(m) \) time

Amortized analysis

- Analyze the time complexity over the entire “lifespan” of the algorithm
- Some operations that cost more will be “covered” by many other operations taking less

Lists and dictionary ADT...

- How to maintain a dictionary using (linked) lists?
- Is \( k \) in \( D \) ?
  - go through all elements \( d \) of \( D \), test if \( d = k \) \( O(n) \)
  - If sorted: \( d = \text{first}(D) \); while (\( d < k \) ) \( d = \text{next}(D); \)
  - on average \( n/2 \) tests ...
- \( \text{Add}(k,D) \Rightarrow \text{insert}(k,D) = O(1) \) or \( O(n) \) – test for uniqueness

Array based sorted list

- is \( d \) in \( D \) ?
- Binary search in \( D \)

```
Binary search – recursive

BinarySearch(A[0..N-1], value, low, high)
{
    if (high < low)
        return -1 // not found
    mid = low + ((high - low) / 2) // Note: not (low + high) / 2 !!
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid // found
}
```

```
Binary search – iterative

BinarySearch(A[0..N-1], value)
{
    low = 0; high = N - 1;
    while (low <= high) {
        mid = low + ((high - low) / 2) // Note: not (low + high) / 2 !!
        if (A[mid] > value)
            high = mid - 1
        else if (A[mid] < value)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}
```
**Work performed**

- $x \leftrightarrow A[18] \ ? \ <$
- $x \leftrightarrow A[9] \ ? \ >$
- $x \leftrightarrow A[13] \ ? \ ==$

$\mathbf{O}(\lg n)$

**Sorting**

- given a list, arrange values so that $L[1] \leq L[2] \leq \ldots \leq L[n]$
- $n$ elements $\Rightarrow n!$ possible orderings
- One test $L[i] \leq L[j]$ can divide $n!$ to 2
  - Make a binary tree and calculate the depth
- $\log(n!) = \Omega(n \log n)$
  - Hence, lower bound for sorting is $\Omega(n \log n)$
    -- using comparisons...

**Decision-tree example**

Sort $(a_1, a_2, a_3) = (9, 4, 6)$:

```
  1
 / \
2 3
 / \
1 2
 / \
3 2
 / \
1 3
```

Each leaf contains a permutation $(\pi(1), \pi(2), \ldots, \pi(n))$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ has been established.

**Lower bound for decision-tree sorting**

**Theorem.** Any decision tree that can sort $n$ elements must have height $\Omega(n \lg n)$.

**Proof.** The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-$h$ binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$\therefore \ h \geq \lg(n!)$

$\geq \lg (n/e)^n$  (Stirling’s formula)

$= n \lg n - n \lg e$

$= \Omega(n \lg n)$.

**Decision tree model**

- $n!$ orderings (leaves)
- Height of such tree?

**Proof: $\log(n!) = \Omega(n \log n)$**

- $\log(n!) = \log n + \log(n-1) + \log(n-2) + \ldots + \log(1)$
  
  $\geq n/2 \times \log(n/2)$

  **Half of elements are larger than log(n/2)**

  $= \Omega(n \log n)$
The divide-and-conquer design paradigm

1. Divide the problem (instance) into subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine subproblem solutions.

\[
\lceil \log_2 n! \rceil \geq \sum_{i=1}^{n} \log_2 i \\
\geq \sum_{i=1}^{n/2} \log_2 i \\
\geq \frac{n/2 \log_2 n}{2} \\
= \Omega(n \log n).
\]

Merge sort

```plaintext
Merge-Sort(A, p, r)
if p<r
    then q = (p+r)/2
        Merge-Sort(A, p, q)  // floor
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

It was invented by John von Neumann in 1945.

Example

- Applying the merge sort algorithm:

```
A = [5, 2, 8, 7, 1, 3, 4, 6]
B = [9, 6, 5, 8, 7, 2, 4, 3]
```

Merge of two lists: \(\Theta(n)\)

A, B – lists to be merged
L = new list; // empty
while( A not empty and B not empty )
    if A.first() <= B.first()
        then append( L, A.first() ); A = rest(A)
        else append( L, B.first() ); B = rest(B)
append( L, A); // all remaining elements of A
append( L, B ); // all remaining elements of B
return L

Wikipedia / viz.
Run-time Analysis of Merge Sort

- Thus, the time required to sort an array of size $n > 1$ is:
  - the time required to sort the first half,
  - the time required to sort the second half, and
  - the time required to merge the two lists
- That is:
  $$T(n) = \begin{cases} \Theta(l) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Merge sort

- Worst case, average case, best case ...
  $\Theta(n \log n)$
- Common wisdom:
  - Requires additional space for merging (in case of arrays)
- Homework*: develop in-place merge of two lists implemented in arrays /compare speed/

Quicksort

- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).

Divide and conquer

Quick sort an $n$-element array:

1. **Divide:** Partition the array into two subarrays around a pivot $x$ such that elements in lower subarray $\leq x$ elements in upper subarray $\geq x$
2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.
   - Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

```plaintext
QUICKSORT(A, p, r)
if p < r
    then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q−1)
        QUICKSORT(A, q + 1, r)
Initial call: QUICKSORT(A, 1, n)
```
Partitioning subroutine

```
Partition(A, p, q) = A[p..q]

x ← A[p]

for j ← p + 1 to q
  do if A[j] ≤ x
     then i ← i + 1

Invariant: x ≤ i ≤ j ≤ q

Running time = O(n) for n elements.
```

Partitioning version 2

```
pivot = A[R];
i = L; j = R - 1;

while (i < j)
  while (A[i] < pivot) i++;
  if (i < j) swap(A[i], A[j]); i++;
  j--;

A[R] = A[i];
A[i] = pivot;
return i;
```

Wikipedia / “video”

```
Worst-case of quicksort

• Input sorted or reverse sorted.
• Partition around min or max element.
• One side of partition always has no elements.

T(n) = T(0) + T(n - 1) + Θ(n)
     = Θ(1) + T(n - 1) + Θ(n)
     = T(n - 1) + Θ(n)
     = Θ(n²) (arithmetic series)
```

Best-case analysis

(For intuition only!)

If we’re lucky, Partition splits the array evenly:

\[ T(n) = 2T(n/2) + \Theta(n) \]

= Θ(n lg n) (same as merge sort)

What if the split is always \( \frac{1}{10} \cdot \frac{9}{10} \)?

\[ T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) \]

What is the solution to this recurrence?

Analysis of “almost-best” case

```
T\left(\frac{n}{10}\right) \quad T\left(\frac{n}{10}\right)
```

14
Choice of pivot in Quicksort

- Select median of three ...
- Select random – opponent can not choose the winning strategy against you!

Randomized quicksort

**Idea:** Partition around a random element.
- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Quick sort in practice

- Quick sort is a great general-purpose sorting algorithm.
- Quick sort is typically over twice as fast as merge sort.
- Quick sort can benefit substantially from code tuning.
- Quick sort behaves well even with caching and virtual memory.
Randomized quicksort analysis

\[ T(n) = \begin{cases} 
T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\
T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\
\vdots & \\
T(n) + 1 + \Theta(n) & \text{if } n-1 : 0 \text{ split}, 
\end{cases} \]

Analysis (continued)

\[ T(n) = \begin{cases} 
T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\
T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\
\vdots & \\
T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, 
\end{cases} \]

= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))

Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right] \]

= \sum_{k=0}^{n-1} E[X_k] (E[T(k) + T(n-k-1) + \Theta(n)])

Linearity of expectation.

Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right] \]

= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]

Independence of \(X_k\) from other random choices.
Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} (T(k) + T(n-k-1) + \Theta(n)) \right] \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \]

Linearity of expectation; \( E[X_i] = 1/n \).

Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} (T(k) + T(n-k-1) + \Theta(n)) \right] \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \Theta(n) \quad \text{Summations have identical terms.} \]

Hairy recurrence

\[ E[T(n)] = 2 \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

(The \( k = 0, 1 \) terms can be absorbed in the \( \Theta(n) \).)

**Prove:** \( E[T(n)] \leq an \log n \) for constant \( a > 0 \).

• Choose \( a \) large enough so that \( an \log n \) dominates \( E[T(n)] \) for sufficiently small \( n \geq 2 \).

**Use fact:** \( \sum_{k=2}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{2} n^2 \) (exercise).

Substitution method

\[ E[T(n)] \leq \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

\[ = \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n) \]

\[ = an \log n - \left( \frac{an}{4} - \Theta(n) \right) \]

\[ \leq an \log n, \]

if \( a \) is chosen large enough so that \( an/4 \) dominates the \( \Theta(n) \).

Alternative materials

• Quicksort average case analysis
  
  [http://eid.ee/10z](http://eid.ee/10z)

  [https://course.stanford.edu/.../quick/space/9120.05.html](https://course.stanford.edu/.../quick/space/9120.05.html)

• [http://eid.ee/10y](http://eid.ee/10y) - MIT Open Courseware - Asymptotic notation, Recurrences, Substitution Master Method

The master method

The master method applies to recurrences of the form

\[ T(n) = a \, T(n/b) + f(n), \]

where \( a \geq 1, b > 1 \), and \( f \) is asymptotically positive.
Three common cases

Compare \( f(n) \) with \( n^{\log_{b}a} \):

1. \( f(n) = \Theta(n^{\log_{b}a}) \) for some constant \( c > 0 \).
   - \( f(n) \) grows polynomially slower than \( n^{\log_{b}a} \) (by an \( n^c \) factor).
   \[ \text{Solution: } T(n) = \Theta(n^{\log_{b}a}). \]

Three common cases (cont.)

Compare \( f(n) \) with \( n^{\log_{b}a} \):

3. \( f(n) = \Theta(n^{\log_{b}a} + \varepsilon) \) for some constant \( \varepsilon > 0 \).
   - \( f(n) \) grows polynomially faster than \( n^{\log_{b}a} \) (by an \( n^c \) factor),
   - and \( f(n) \) satisfies the regularity condition that \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \).
   \[ \text{Solution: } T(n) = \Theta(f(n)). \]

Examples

\[ T(n) = 4T(n/2) + n^3 \]
\[ a = 4, \ b = 2 \Rightarrow n^{\log_{b}a} = n^2; f(n) = n^3. \]

**Case 3.** \( f(n) = \Omega(n^{\log_{b}a} + \varepsilon) \) for \( \varepsilon = 1 \)

and \( 4(n/2)^3 \leq cn^3 \) (reg. cond.) for \( c = 1/2 \).

\[ \therefore T(n) = \Theta(n^3). \]

Examples

\[ T(n) = 4T(n/2) + n^3 \]
\[ a = 4, \ b = 2 \Rightarrow n^{\log_{b}a} = n^2; f(n) = n^3. \]

**Case 3.** \( f(n) = \Omega(n^{\log_{b}a} + \varepsilon) \) for \( \varepsilon = 1 \)

and \( 4(n/2)^3 \leq cn^3 \) (reg. cond.) for \( c = 1/2 \).

\[ \therefore T(n) = \Theta(n^3). \]

Examples

\[ T(n) = 4T(n/2) + n^{\log_{b}a} \ln n \]
\[ a = 4, \ b = 2 \Rightarrow n^{\log_{b}a} = n^2; f(n) = n^2 \ln n. \]

Master method does not apply. In particular, for every constant \( \varepsilon > 0 \), we have \( n^\varepsilon = \Omega(\ln n). \)
Idea of master theorem

Recursion tree:

\[ f(n) \]
\[ a \]
\[ f(n/b) \]
\[ \cdots \]
\[ f(n/b^k) \]
\[ f(n/b^{k+1}) \]
\[ T(1) \]

\[ h = \log_b a \]

\#leaves = \[ a^h = a^{\log_b n} = n^{\log_b a} \]

\[ T(1) \]

\[ \Theta(n^{\log_b a}) \]

CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

CASE 2: (k = 0) The weight is approximately the same on each of the \[ \log_b n \] levels.

CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.
Back to sorting

We can sort in $O(n \log n)$

- Is that the best we can do?
- Remember: using comparisons $<$, $>$, $\leq$, $\geq$ we can not do better than $O(n \log n)$

How fast can we sort $n$ integers?

- E.g. sort people by year of birth?
- Sort people by their sex?

Sorting in linear time

**Counting sort:** No comparisons between elements.

- **Input:** $A[1..n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output:** $B[1..n]$, sorted.
- **Auxiliary storage:** $C[1..k]$.

**Counting sort**

```plaintext
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1 ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i−1] ▷ C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
```

**Loop 1**

```
A:  4  1  3  4  3
B:    □ □ □ □ □
C:  0  0  0  0
```
### Loop 2

**A:**

```
1 2 3 4 5
4 1 3 4 3
```

**B:**

```
```

**C:**

```
1 2 3 4
1 0 2 2
```

- for $j \leftarrow 1$ to $n$
- do $C[A[j]] \leftarrow C[A[j]] + 1$  \( \triangleright C[i] = |\{\text{key = i}\}| \)

### Loop 3

**A:**

```
1 2 3 4 5
4 1 3 4 3
```

**B:**

```
```

**C:**

```
1 2 3 4
1 0 2 2
```

- for $i \leftarrow 2$ to $k$
- do $C[i] \leftarrow C[i] + C[i-1]$  \( \triangleright C[i] = |\{\text{key \leq i}\}| \)

### Loop 4

**A:**

```
1 2 3 4 5
4 1 3 4 3
```

**B:**

```
3 4
```

**C:**

```
1 2 3 4
1 1 2 5
```

- for $j \leftarrow n$ downto 1
- do $B[C[A[j]]] \leftarrow A[j]$
- \( C[A[j]] \leftarrow C[A[j]] - 1 \)

### Analysis

\( \Theta(k) \)

- for $i \leftarrow 1$ to $k$
- do $C[i] \leftarrow 0$

\( \Theta(n) \)

- for $j \leftarrow 1$ to $n$
- do $C[A[j]] \leftarrow C[A[j]] + 1$

\( \Theta(k) \)

- for $i \leftarrow 2$ to $k$
- do $C[i] \leftarrow C[i] + C[i-1]$

\( \Theta(n) \)

- for $j \leftarrow n$ downto 1
- do $B[C[A[j]]] \leftarrow A[j]$

\( \Theta(n + k) \)

### Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.
- But, sorting takes $\Omega(n \log n)$ time!
- Where’s the fallacy?

**Answer:**

- **Comparison sorting** takes $\Omega(n \log n)$ time.
- Counting sort is not a **comparison sort**.
- In fact, not a single comparison between elements occurs!

### Stable sorting

Counting sort is a **stable** sort: it preserves the input order among equal elements.

**A:**

```
4 1 3 4 3
```

**B:**

```
1 3 4 4
```

**Exercise:** What other sorts have this property?
Radix sort

• **Origin:** Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix 4.)
• Digit-by-digit sort.
• Hollerith’s original (bad) idea: sort on most-significant digit first.
• Good idea: Sort on *least-significant digit first* with auxiliary stable sort.

**Operation of radix sort**

329  720  720  329
457  355  329  355
657  436  436  436
839  457  839  457
436  657  355  657
720  329  457  720
355  839  657  839

Correctness of radix sort

**Induction on digit position**

• Assume that the numbers are sorted by their low-order \( t-1 \) digits.
• Sort on digit \( t \)
  • Two numbers that differ in digit \( t \) are correctly sorted.

Analysis of radix sort

• Assume counting sort is the auxiliary stable sort.
• Sort \( n \) computer words of \( b \) bits each.
• Each word can be viewed as having \( b/r \) base-2\(^r\) digits.

**Example:** 32-bit word

\[ \begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array} \]

\( r = 8 \Rightarrow b/r = 4 \) passes of counting sort on base-2\(^4\) digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-2\(^2\) digits.

**How many passes should we make?**
Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort $n$ numbers in the range from 0 to $k - 1$. If each $b$-bit word is broken into $r$-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are $b/r$ passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r} \left(n + 2^r\right)\right).$$

Choose $r$ to minimize $T(n, b)$:
- Increasing $r$ means fewer passes, but as $r \gg \log n$, the time grows exponentially.

Choosing $r$ by differentiating and setting to 0.

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don’t want $2^r \gg n$, and there’s no harm asymptotically in choosing $r$ as large as possible subject to this constraint.

Choosing $r = \log n$ implies $T(n, b) = \Theta(bn/\log n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \log n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):
- At most 3 passes when sorting $\geq 2000$ numbers.
- Merge sort and quicksort do at least $\lceil \log 2000 \rceil = 11$ passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c e c c e c e c e c e c e c c e c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c e c c e c e c e c e c e c c e c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c c c c c c c c c c c c c c c c c c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c c c c c c c c c c c c c c c c c c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c c c c c c c c c c c c c c c c c c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c c c c c c c c c c c c c c c c c c c
   d d d d d d d d d d d d d d d d d d d d
```

Radix sort using lists (stable)

```
1. a blub blue blue blue blue blue
   b blue blue blue blue blue blue
   c c c c c c c c c c c c c c c c c c c c
   d d d d d d d d d d d d d d d d d d d d
```
Why not from left to right?

- Swap ‘0’ with first ‘1’
- Idea 1: recursively sort first and second half
  — Exercise?

Bitwise sort left to right

- Idea 2:
  - swap elements only if the prefixes match...
  - For all bits from most significant
  - advance when 0
  - when 1 → look for next 0
  - if prefix matches, swap
  - otherwise keep advancing on 0’s and look for next 1

Bitwise left to right sort

/* Historical sorting – was used in Univ. of Tartu using assembler… */
/* C implementation – Jaak Vilo, 1989 */

void bitwisesort (SORTTYPE *ARRAY, int size)
{
  int i, j, tmp, nrbits;
  register SORTTYPE mask, curbit, group;

  nrbits = sizeof(SORTTYPE) * 8;
  curbit = 1 << (nrbits-1); /* set most significant bit 1 */
  mask = 0; /* mask of the already sorted area */

  do { /* For each bit */
    nrbits = 0;
    curbit = 1 << (nrbits-1); /* set most significant bit 1 */
    mask = 0; /* mask of the already sorted area */

    for (i = 0; i < size; i++)
    {
      if (! (ARRAY[i] & curbit)) /* bit is 0 – need to swap with previous location of 1, if 1 = 0 */
      {
        tmp = ARRAY[i];
        ARRAY[i] = ARRAY[i] & mask;
        ARRAY[i] |= (curbit & (i < size)); /* advance white bit to 0 */
        if (i < size) goto array_end; /* reached end of array */
        group = ARRAY[i] & mask;
      }
    }

    /* save current prefix snapshot */

    new_mask:
    mask |= curbit; /* area under mask is now sorted */
    curbit >>= 1; /* next bit */
    if (curbit) goto new_mask; /* until all bits have been sorted… */

    array_end:

  } while(curbit);
}

Jaak Vilo, Univ. of Tartu

Bitwise from left to right

0101100 0101100 0101000 0101100
1001001 1001100 1001000 0101000
1011000 1111000 0101000 0101000
0101001 1001010 1001001 1001001
0101000 0101000 1111000 1111000
0010000 0010000 0010000 0010000
1001001 1001001

- Swap ‘0’ with first ‘1’

Jaak Vilo, Univ. of Tartu

Bucket sort

- Assume uniform distribution
- Allocate O(n) buckets
- Assign each value to pre-assigned bucket
Sort small buckets with insertion sort

- The sort input records must be 100 bytes in length, with the first 10 bytes being a random key.
- Minutesort – max amount sorted in 1 minute
  - 116GB in 58.7 sec (Jim Wyllie, IBM Research)
  - 40-node 80-Titanium cluster, SAN array of 2,520 disks
- 2009, 500 GB Hadoop, 1406 nodes (2 Quadcore Xeons, 8 GB memory, 4 SATA)
  Owen O’Malley and Arun Murthy
  Yahoo Inc.
- Performance / Price Sort and PennySort

http://sortbenchmark.org/

- Daytoma
- Indy
- Penny
- Minutesort
- TeraByte
- Jungle

**Year 2010 Results**

- **Daytoma**
  - 1.41 GB in 3 minutes (99 GB in 2,000 sequenced)
  - 3.66 GB in 12 minutes (189 GB in 6,000 sequenced)

- **Indy**
  - 3 GB in 2 hours (156 GB in 8,000 sequenced)

- **Penny**
  - 2 MB in 30 seconds (13 MB in 1,000 sequenced)

- **Minutesort**
  - 240 GB in 3 minutes (1,400 GB in 6,000 sequenced)
  - 790 GB in 10 minutes (4,700 GB in 24,000 sequenced)

- **TeraByte**
  - 1.4 TB in 20 hours (10.5 TB in 90,000 sequenced)

- **Jungle**
  - 10 GB in 2 hours (60 GB in 120,000 sequenced)

- **Daytona**
  - 1.4 TB in 20 hours (10.5 TB in 90,000 sequenced)

**Top Results**

- **TrifonSort**
  - 300 GB in 1 hour (1,800 GB in 12,000 sequenced)

- **Sort**
  - 1.4 TB in 1 hour (10.5 TB in 90,000 sequenced)

- **FastSort**
  - 1.4 TB in 1 hour (10.5 TB in 90,000 sequenced)

- **FastSort2**
  - 1.4 TB in 1 hour (10.5 TB in 90,000 sequenced)

- **FastSort3**
  - 1.4 TB in 1 hour (10.5 TB in 90,000 sequenced)

- **FastSort4**
  - 1.4 TB in 1 hour (10.5 TB in 90,000 sequenced)

**Year 2009 Results**

**New rules for GraySort:**

- The input file size is now minimum 100TB or 1T records. Entries with larger input sizes also qualify.
- The winner will have the fastest SortedRecs/Min.
- We now provide a new input generator that works in parallel and generates binary data. See below.
- For the Daytona category, we have two new requirements. (1) The sort must run continuously (repeatedly) for a minimum 1 hour. (This is a minimum reliability requirement). (2) The system cannot overwrite the input file.
- New hardware used must be off-the-shelf and unmodified.
- For Daytona cluster sorts, when input sampling is used to determine the output partition boundaries, the input sampling must be done evenly across all input partitions.

**Performance / Price**

- Sort and PennySort
**Order statistics**

- Minimum – the smallest value
- Maximum – the largest value
- In general i’th value.
- Find the median of the values in the array
- Median in sorted array A:
  - n is odd \( A[(n+1)/2] \)
  - n is even – \( A[(n+1)/2] \) or \( A[(n+1)/2] \)

**Input:** A set A of n numbers and i, \( 1 \leq i \leq n \)

**Output:** x from A that is larger than exactly i-1 elements of A

**Q: Find i’th value in unsorted data**

A. \( O(n) \)
B. \( O(n \log \log n) \)
C. \( O(n \log n) \)
D. \( O(n \log^2 n) \)

**Minimum**

\[
\begin{align*}
\text{Minimum}(A) \\
1 \quad & \text{min} = A[1] \\
2 \quad & \text{for } i = 2 \text{ to length}(A) \\
3 \quad & \text{if } \text{min} > A[i] \\
4 \quad & \text{then } \text{min} = A[i] \\
5 \quad & \text{return } \text{min}
\end{align*}
\]

n-1 comparisons.

**Min and max together**

- compare every two elements \( A[i], A[i+1] \)
- Compare larger against current max
- Smaller against current min
- \( 3 \lceil n / 2 \rceil \)

**Selection in expected \( O(n) \)**

\[
\begin{align*}
\text{Randomised-select}(A, p, r, i) \\
\text{if } p=r \text{ then return } A[p] \\
q &= \text{Randomised-Partition}(A, p, r) \\
k &= q - p + 1 \quad \text{// nr of elements in subarr} \\
\text{if } i \leq k \\
\quad & \text{then return } \text{Randomised-Partition}(A, p, q, i) \\
\text{else return } & \text{Randomised-Partition}(A, q+1, r, i-k)
\end{align*}
\]
Conclusion

- Sorting in general $O(n \log n)$
- Quicksort is rather good
- Linear time sorting is achievable when one does not assume only direct comparisons
- Find $i$’th value – expected $O(n)$
- Find $i$’th value: worst case $O(n)$ – see CLRS

Lists: Array

```
3 6 7 5 2
```

Ok…

- lists – a versatile data structure for various purposes
- Sorting – a typical algorithm (many ways)
- Which sorting methods for array/list?

- Array: most of the important (e.g. update) tasks seem to be $O(n)$, which is bad

Q: search for a value $X$ in linked list?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
Can we search faster in linked lists?

• Why sort linked lists if search anyway O(n)?

• Linked lists:
  – what is the “mid-point” of any sublist?
  – Therefore, binary search can not be used...

• Or can it?

Skip lists

• Build several lists at different “skip” steps

• O(n) list
• Level 1: ~ n/2
• Level 2: ~ n/4
• ...
• Level log n ~ 2-3 elements...

Skip List

typedef struct nodeStructure *node;
typedef struct nodeStructure{
  keyType key;
  valueType value;
  node forward[1]; /* variable sized array of forward pointers */
};
Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution.
- It contains statements of the type
  \[ b \leftarrow \text{random}() \]
  
  if \( b = 0 \)
  do A
  else \( b = 1 \)
  do B
- Its running time depends on the outcomes of the coin tosses.
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent

The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give “heads!”)

We use a randomized algorithm to insert items into a skip list.

Search

- We search for a key \( x \) in a skip list as follows:
  - We start at the first position of the top list
  - At the current position \( p \), we compare \( x \) with \( y = \text{key}(p) \)
    - \( x = y \): we return element \((p)\)
    - \( x < y \): we “scan forward”
    - \( x > y \): we “drop down”
  - If we try to drop down past the bottom list, we return NO_SEARCH_KEY
- Example: search for 78

Insertion

- To insert an item \((x, v)\) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with \( i \) the number of times the coin came up heads
  - If \( i < k \), we add to the skip list new lists \( S_0, \ldots, S_{i-1} \), each containing only the two special keys
  - We search for \( x \) in the skip list and find the positions \( p_0, p_1, \ldots, p_i \) of the items with largest key less than \( x \) in each list \( S_0, S_1, \ldots, S_{i-1} \)
  - For \( j = 0, \ldots, \), we insert item \((x, v)\) into list \( S_j \) after position \( p_j \)
- Example: insert key 15, with \( k = 2 \)

Deletion

- To remove an item with key \( x \) from a skip list, we proceed as follows:
  - We search for \( x \) in the skip list and find the positions \( p_0, p_1, \ldots, p_i \) of the items with key \( x \), where position \( p_i \) is in list \( S_i \)
  - We remove positions \( p_0, p_1, \ldots, p_i \) from the lists \( S_0, S_1, \ldots, S_i \)
  - We remove all but one list containing only the two special keys
- Example: remove key 34

Implementation v2

- We can implement a skip list with quad-nodes:
  - A quad-node stores:
    - item
    - link to the node before
    - link to the node after
    - link to the node below
    - link to the node above
  - Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them.
Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm.
- We use the following two basic probabilistic facts:
  
  **Fact 1:** The probability of getting $i$ consecutive heads when flipping a coin is $1/2^i$.
  
  **Fact 2:** If each of $n$ items is present in a set with probability $p$, the expected size of the set is $np$.

- Consider a skip list with $n$ items:
  
  - By Fact 1, we insert an item in list $S_i$ with probability $1/2^i$.
  - By Fact 2, the expected size of list $S_i$ is $n/2^i$.

  The expected number of nodes used by the skip list is

  $$\sum_{i=1}^{\infty} \frac{n}{2^i} = n \cdot \frac{1}{1 - \frac{1}{2}} = 2n.$$

  Thus, the expected space usage of a skip list with $n$ items is $O(n)$.

Search and Update Times

- The search time in a skip list is proportional to:
  
  - the number of drop-down steps, plus
  - the number of scan-forward steps.

- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability.

- To analyze the scan-forward steps, we use yet another probabilistic fact:
  
  **Fact 4:** The expected number of coin tosses required in order to get tails is $2$.

- When we scan forward in a list, the destination key does not belong to a higher list.

- A scan-forward step is associated with a former coin toss that gave tails.

- By Fact 4, in each list the expected number of scan-forward steps is $2$.

- Thus, the expected number of scan-forward steps is $O(\log n)$.

- We conclude that a search in a skip list takes $O(\log n)$ expected time.

- The analysis of insertion and deletion gives similar results.

Height

- The running time of the search algorithm is affected by the height $h$ of the skip list.

- We show that with high probability, a skip list with $n$ items has height $O(\log n)$.

- We use the following additional probabilistic fact:
  
  **Fact 3:** If each of $n$ events has probability $p$, the probability that at least one event occurs is at most $np$.

- Consider a skip list with $n$ items:
  
  - By Fact 1, we insert an item in list $S_i$ with probability $1/2^i$.
  - By Fact 3, the probability that list $S_i$ has at least one item is at most $n/2^i$.

- By picking $i = \log n$, we have that the probability that $S_{\log n}$ has at least one item is at most $n^{\log n}/2^{\log n} = n/2 = 1/2$.

- Thus, a skip list with $n$ items has height at most $\log n$ with probability at least $1 - 1/2^n$.

Summary

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm.

- In a skip list with $n$ items:
  
  - The expected space used is $O(n \log n)$.
  - The expected search, insertion and deletion time is $O(\log n)$.

- By picking $i = \log n$, we have that the probability that $S_{\log n}$ has at least one item is at most $n/2^{\log n} = n/2 = 1/2$.

- Thus, a skip list with $n$ items has height at most $\log n$ with probability at least $1 - 1/2^n$.

- Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability.

- Skip lists are fast and simple to implement in practice.

Conclusions

- Abstract data types hide implementations.

- Important is the functionality of the ADT.

- **Data structures and algorithms** determine the speed of the operations on data.

- Linear data structures provide good versatility.

- Sorting – a most typical need/algorithm.

- Sorting in $O(n \log n)$: Merge Sort, Quicksort.

- Solving Recurrences – means to analyse.

- Skip lists – log $n$ randomised data structure.