Lists: Array

L = int[MAX_SIZE]
L[2]=7

Linear, sequential, list ...

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
- Stack (LIFO)
- Queue (FIFO)
- Queue (double-ended)
- Priority queue (fetch highest-value 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

Some data structures for Dictionary ADT

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

Lists: Array

- Insert 8 after L[2]
- Delete last
Lists: Array

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

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

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

Linked lists

Linked lists: add/delete

Operations

- Array indexed from 0 to $n - 1$:

<table>
<thead>
<tr>
<th>$k = 1$</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>$O(n)$</td>
<td>$O(n)$</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>$k = 1$</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>$O(n)$</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>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

under the assumption we have a pointer to the $k$th node, $O(k)$ otherwise
Improving Run-Time Efficiency

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

Singly-linked list:

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

Doubly-linked list:

• Comparing the tables:

<table>
<thead>
<tr>
<th></th>
<th>( k = 1 )</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(n) )</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>
<td>( O(n) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

* under the assumption we have a pointer to the \( k \)th node, \( O(1) \) otherwise

Introduction to linked lists:

**definition**

• Consider the following struct definition

```c
struct node
{
    string word;
    int num;
    node *next;  // pointer for the next node
};
```

```c
node *p = new node;
```

node *p = new node;

**inserting a node**

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

```c
node *q;
q = new node;
```

**adding a new node**

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

```c
node *q;
q = new node;
```

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

```c
node *q;
q = new node;
```
Introduction to linked lists

- node *p, *q;
- 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;

Pointers

- 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:
  http://cslibrary.stanford.edu/106/

- C++ Memory Management : What is the difference between malloc/free and new/delete?
To test and understand – use int’s

• 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

Maintaining list of free objects

Multiple lists, single free list

Hack: allocate more arrays …

XOR linked lists are a data structure used in computer programming. They take advantage of the bitwise exclusive disjunction (XOR) operation, here denoted by ⊕, to decrease storage requirements for doubly-linked lists. An ordinary doubly-linked list stores addresses of the previous and next list items in each list node, requiring two address fields.
Queue
(basic idea, does not contain all controls!)

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

First = List[F]
Last = List[L-1]
Full: return (L==MAX_SIZE)
Empty: F<0 or F>=L

• Queue
  • `enqueue(x)` - add to end
  • `dequeue()` - fetch from beginning
  • FIFO – First In First Out
  • O(1) in all reasonable cases 😊

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

<table>
<thead>
<tr>
<th>L</th>
<th>F</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

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

• Circular 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 be?

\[ \begin{array}{c}
3 & 6 & 7 & 5 \\
3 & 6 & 7 & 5 & 2 & 1
\end{array} \]

• When full, dynamically allocate bigger table, 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 +

• \(n/2 + n/4 + n/8 \ldots\) copy-ing

• Total: \(O(n)\) effort!

• when \(n=32\) -> 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...

• 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( \(d<\text{k}\) ) \(d=\text{next}(D)\);
  – on average \(n/2\) tests...

• Add(\(k,D\)) => 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 <=> A[18] ? <
• x <=> A[9] ? >
• x <=> A[13] ? ==

• O(lg n)

Sorting

• given a list, arrange values so that
• n elements => n! possible orderings
• One test L[i] <= L[j] can divide n! to 2
  − Make a binary tree and calculate the depth
• log( n! ) = Ω ( n log n )
• Hence, lower bound for sorting is Ω ( n log n )
  − using comparisons...

Proof: log(n!) = Ω ( n log n )

• log( n! ) = log n + log (n-1) + log(n-2) + ... log(1)
  >= n/2 * log( n/2 )
  = Ω ( n log n )

Decision-tree example

Sort (a1, a2, a3)
= (9, 4, 6):

Each leaf contains a permutation (π(1), π(2), ..., π(n)) to indicate the ordering a_{π(1)} ≤ a_{π(2)} ≤ ... ≤ a_{π(n)} has been established.
Decision tree model

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

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

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.

Merge sort

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

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:

\[
T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
2T(\frac{n}{2}) + \Theta(n) & n > 1 
\end{cases}
\]

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

QuickSort an \(n\)-element array:

1. **Divide:** Partition the array into two subarrays around a pivot \(x\) such that elements in lower subarray \(\leq x\) are elements in upper subarray \(\geq x\).

2. **Conquer:** Recursively sort the two subarrays.

3. **Combine:** Trivial.

   **Key:** Linear-time partitioning subroutine.

Pseudocode for quicksort

```c
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)
```

Partioning subroutine

```c
PARTITION(A, p, q)
    x ← A[p]
    pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
            then i ← i + 1
    return i
```

Partitioning version 2

```
pivot = A[R]; //
i=L; j=R-1;
while( i<j )
    while ( A[i] < pivot ) i++ ; // will stop at pivot latest
    while ( i<j and A[j] >= pivot ) j--;
A[R]=A[i];
A[i]=pivot;
return i;
```
**Worst-case of quicksort**

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

\[
T(n) = T(0) + T(n-1) + \Theta(n) \\
= \Theta(1) + T(n-1) + \Theta(n) \\
= \Theta(n^2) \quad \text{(arithmetic series)}
\]

**Best-case analysis**

*For intuition only!*

If we’re lucky, **PARTITION** splits the array evenly:

\[
T(n) = 2T(n/2) + \Theta(n) \\
= \Theta(n \log n) \quad \text{(same as sort)}
\]

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

\[
T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + \Theta(n)
\]

What is the solution to this recurrence?

**Analysis of “almost-best” case**

\[
T(\frac{1}{10}n) \quad T(\frac{9}{10}n)
\]

**More intuition**

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ...,

\[
I(n) = 2U(n/2) + \Theta(n) \quad \text{lucky} \\
U(n) = I(n-1) + \Theta(n) \quad \text{unlucky}
\]

Solving:

\[
I(n) = 2(I(n/2) + \Theta(n/2)) + \Theta(n) \\
= 2I(n/2 - 1) + \Theta(n) \\
= \Theta(n \log n) \quad \text{Lucky!}
\]

How can we make sure we are usually lucky?

**Choice of pivot**

- Select median of three ...
- Select random – opponent cannot 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.

---

Random pivot

Select pivot randomly from the region (blue) and swap with last position

Select pivot as a median of 3 [or more] random values from region

Apply non-recursive sort for array less than 10-20

---

Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size $n$, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable $X_k$ =

- 1 if PARTITION generates a $k : n-k-1$ split,
- 0 otherwise.

$E[X_k] = \Pr[X_k = 1] = 1/n$, since all splits are equally likely, assuming elements are distinct.

---

Analysis (continued)

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

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

---

Calculating expectation

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

Take expectations of both sides.

---

Calculating expectation

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

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

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

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

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

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

Calculating expectation

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

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

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

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

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

• Choose \( a \) large enough so that \( an \lg n \)

dominates \( E[T(n)] \) for sufficiently small \( n \geq 2 \).

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

Substitution method

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

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

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

\[
\leq an \lg n
\]

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

QuickSort in practice

• QuickSort is a great general-purpose sorting algorithm.

• QuickSort is typically over twice as fast as merge sort.

• QuickSort can benefit substantially from code tuning.

• QuickSort behaves well even with caching and virtual memory.
We can sort in $O(n \log n)$

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

How fast can we sort $n$ integers?

**Sorting in linear time**

*Counting sort:* No comparisons between elements.
- **Input:** $A[1..n]$, where $A[j] \in \{1, 2, ..., k\}$.
- **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**

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for $i ← 1$ to $k$
    do $C[i] ← 0$

**Loop 2**

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for $j ← 1$ to $n$
    do $C[A[j]] ← C[A[j]] + 1$  ▷ $C[i] = |\{key = i\}|$
20.2.2011

**Loop 3**

\[
\begin{array}{cccc}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
C: & 1 & 0 & 2 & 2 \\
\end{array}
\]

\[
\text{for } i \leftarrow 2 \text{ to } k \\
\text{do } C[i] \leftarrow C[i] + C[i-1] \quad \text{▷ } C[i] = |\{\text{key } \leq i\}|
\]

**Loop 4**

\[
\begin{array}{cccc}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & 3 & 4 & & \\
C: & 1 & 1 & 2 & 5 \\
\end{array}
\]

\[
\text{for } j \leftarrow n \text{ downto } 1 \\
\text{do } B[C[A[j]]] \leftarrow A[j] \\
C[A[j]] \leftarrow C[A[j]] - 1
\]

**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 \(B[C[A[j]]] \leftarrow A[j]\)
  - do \(C[A[j]] \leftarrow C[A[j]] - 1\)

- \(\Theta(n + k)\)

**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.

\[
\begin{array}{cccc}
A: & 4 & 1 & 3 & 4 & 3 \\
B: & 1 & 3 & 3 & 4 & 4 \\
\end{array}
\]

**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.
Radix sort

Radix-Sort(A,d)
1. for i = 1 to d
2. do use a stable sort to sort A on digit i

Operation of radix sort

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\(^8\) digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-2\(^{16}\) 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 \(r \gg \lg n\), the time grows exponentially.
Choosing $r$

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

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 $T(n, b) = \Theta(b n / \log n)$.

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

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

- Idea2:
  - 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

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

void bitwisesort(SORTTYPE *ARRAY, int size)
{

/* Set most significant bit 1 */

int i, j, tmp, nrbits;

register SORTTYPE mask, curbit, group;

nrbits = sizeof(SORTTYPE) * 8;

curbit = 1 << (nrbits - 1);

/* Save current prefix snapshot */

mask = 0;

group = ARRAY[i] & mask;

j = i;

/* Memorize location of 1 */

for(;;)
{
  if (++i >= size) goto array_end;

  /* Reached end of array */

  if (ARRAY[i] & mask) goto new_mask;

  /* New prefix */

  if (! (ARRAY[i] & curbit))
  {
    tmp = ARRAY[i];
    ARRAY[i] = ARRAY[j];
    ARRAY[j] = tmp;
    j += 1;
  } /* Swap and increase to the next possible 1 */

array_end:

mask = mask | curbit;

/* Area under mask is now sorted */

curbit >>= 1;

/* Until all bits have been sorted… */

}

Bitwise left to right sort

void bitwisesort(SORTTYPE *ARRAY, int size)
{

/* Set most significant bit 1 */

int i, j, tmp, nrbits;

register SORTTYPE mask, curbit, group;

nrbits = sizeof(SORTTYPE) * 8;

curbit = 1 << (nrbits-1);

mask = 0;

Jaak Vilo, Univ. of Tartu

Bitwise from left to right

0010000
0010010
0101000
0101100
1001000
1001001
1111000

• 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

0 1 2 3 4 5 6 7 8 9

0 / .12 .13 .21 .23 .25 .39 / .68 .72 .78 .94

Jaak Vilo, Univ. of Tartu
http://sortbenchmark.org/

- Minutesort – max amount sorted in 1 minute
  - 116GB in 58.7 sec (Jim Wyllie, IBM Research)
  - 40-node 80-Itanium cluster, SAN array of 2,520 disks
- 2009, 500 GB Hadoop 1406 nodes x (2 Quadcore Xeons, 8 GB memory, 4 SATA)
  - Owen O’Malley and Arun Murthy
  - Yahoo Inc.
- Performance / Price Sort and PennySort

**Year 2009 Results**

<table>
<thead>
<tr>
<th></th>
<th>Penny</th>
<th>Indy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>116 GB</td>
<td>116 GB</td>
</tr>
<tr>
<td></td>
<td>58.7 sec</td>
<td>58.7 sec</td>
</tr>
<tr>
<td></td>
<td>40-node</td>
<td>40-node</td>
</tr>
<tr>
<td></td>
<td>80-Itanium</td>
<td>80-Itanium</td>
</tr>
<tr>
<td></td>
<td>cluster,</td>
<td>cluster,</td>
</tr>
<tr>
<td></td>
<td>SAN array of 2,520 disks</td>
<td>SAN array of 2,520 disks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Sort Benchmark**

- http://sortbenchmark.org/
- Sort Benchmark Home Page
- We have a new benchmark called GraySort, in memory of the father of the sort benchmarks, Jim Gray. It replaces TeraByteSort which is now retired.
- The submission deadline is new 15 April 2009.
- New rules for GraySort:
  - The input file size is now minimum ~100TB or 1T records. Entries with larger input sizes also qualify.
  - 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).
  - The system cannot overwrite the input file.

**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 \( A[(n+1)/2] \)}
Minimum

Minimum(A)
1 min = A[1]
2 for i = 2 to length(A)
3 if min > A[i]
4 then min = A[i]
5 return min

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
• \(3\sqrt[4]{n} / 2\)

Selection in expected O(n)

Randomised-select( A, p, r, i )
if p=r then return A[p]
q = Randomised-Partition(A, p, r)
k = q - p + 1 // nr of elements in subarr
if i<= k
then return Randomised-Partition(A, p, q, i)
else return Randomised-Partition(A, q+1, r, i-k)

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

Lists:

<table>
<thead>
<tr>
<th>size</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Insert 8 after L[2]

Delete last

Linked lists

<table>
<thead>
<tr>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Singly linked

Doubly linked
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

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 List

A skip list, introduced by Pagh [Pagh 1999], is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Level 0 of a skip list is a linked list of all nodes in increasing order by key. For each $i$ greater than 0, each node is level $i$ – 1 appears in level $i$ independently with some fixed probability $p$. In a doubly-linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $p n$ pointers per node. The lists at the higher levels act as “reverse lists” that allow the sequence of nodes to be traversed upright. Searching for a node with a particular key involves scanning first in the highest level, and repeatedly dropping down a level whenever it becomes clear that the node is not in the current level. Considering the search path in reverse shows that no more than $O(p n)$ nodes are searched on average per level, giving an average search time of $O(p^{-1/2} n^{1/2})$ with $n$ nodes at level 0. Skip lists have been extensively studied [Pagh 1999; Pagh et al. 1999; Bentley 1992; Kirschbaum and Preparata 1991; Kirschbaum et al. 1999], and because they support no global balancing operations are particularly well in parallel systems [Khemani et al. 1996; Garabi and Hoepegan 1997].

Fig. 1. A skip list with $n = 8$ nodes and $2 = 3$ levels.

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

Skip Lists

- Build several lists at different “skip” steps
- $O(n)$ list
- Level 1: $\sim n/2$
- Level 2: $\sim n/4$
- ...
- Level $\log n \sim 2\cdot3$ elements...
Outline and Reading

- What is a skip list (§3.5)
- Operations
  - Search (§3.5.1)
  - Insertion (§3.5.2)
  - Deletion (§3.5.2)
- Implementation
- Analysis (§3.5.3)
  - Space usage
  - Search and update times

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 \( p \rightarrow \text{keyafter}(p) \)
    - \( x = p \rightarrow \text{keyafter}(p) \): return \( \text{elementafter}(p) \)
    - \( x > p \rightarrow \text{keyafter}(p) \): we "scan forward"
    - \( x < p \rightarrow \text{keyafter}(p) \): we "drop down"
  - If we try to drop down past the bottom list, we return \( \text{NO_SUCH_KEY} \)
- Example: search for 78

Insertion

- To insert an item \( (x, e) \) 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 \( i \leq k \), we add to the skip list new lists \( S_{i-1}, \ldots, S_0 \), 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 \( S_0, S_1, \ldots, S_i \)
  - For \( j = i, \ldots, 0 \), we insert item \( (x, e) \) into list \( S_j \) after position \( p_j \)
- Example: insert key 15, with \( i = 2 \)

Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution:
  - It contains statements of the type
    \[
    b = \text{random()}
    \]
    - if \( b = 0 \) do \( A \) …
    - else \( b = 1 \) do \( B \) …
  - Its running time depends on the outcomes of the coin tosses

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 \( p_0 \) is in list \( S_0 \)
  - 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

What is a Skip List

- A skip list for a set \( S \) of distinct (key, element) items is a series of lists \( S_0, S_1, \ldots, S_k \) such that
  - Each list \( S_i \) contains the special keys \( + \) and \( - \)
  - List \( S_0 \) contains the keys of \( S \) in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., \( S_0 \subseteq S_1 \subseteq \cdots \subseteq S_k \)
  - List \( S_k \) contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT

Outline and Reading
### Implementation

- 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 \( n p \).

### Search and Update Times

- 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.

### 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 \( O(n) \).
  - The expected search, insertion and deletion time is \( O(\log n) \).
- 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