Advanced Algorithmics (4AP)
Linear structures:
Lists, Queues, Stacks, sorting,…

Jaak Vilo
2009 Spring
Lists: Array

\[ L = \text{int}[\text{MAX}_\text{SIZE}] \]

\[ L[2] = 7 \]
Lists: Array

L[2]=7

L[3]=7
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)
Dictionary

• Container of key-element 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

• Ordered
  – Array
  – Sequence (Skip Lists)
  – Binary Search Tree (BST)
  – AVL
  – (2; 4) Trees
  – B-Trees

• Valued
  – Hash Tables
  – Extendible Hashing
Lists: Array

0 1  
3 6 7 5 2  
size  MAX_SIZE-1

Insert 8 after L[2]

0 1  
3 6 7 8 5 2  
size

Delete last

0 1  
3 6 7 8 5 2  
size
Lists: Array

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

Insert 8 after $L[2]$

Delete last
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

Singly linked

Doubly linked
Linked lists: add/delete

[Diagrams showing add and delete operations on linked lists]
### Operations

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

<table>
<thead>
<tr>
<th>Operation</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>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert before or after the $k$th element</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete the $k$th element</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

• **Singly-linked list with head and tail pointers**

<table>
<thead>
<tr>
<th>Operation</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>$O(n)$</td>
<td>$O(1)$</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>

1 under the assumption we have a pointer to the $k$th node, $O(n)$ 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:**
  
  ```
  list_head → A → B → C → ... → Y → Z → ∅
  list_tail
  ```

  **Doubly-linked list:**
  
  ```
  list_head
  ∅ ← A ← B ← C ← ... ← Y ← Z ← ∅
  list_tail
  ```

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

• Comparing the tables:

### Singly-linked list:

<table>
<thead>
<tr>
<th>Action</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>$O(n)$</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>

### Doubly-linked list:

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

$^1$ under the assumption we have a pointer to the $k$th node, $O(n)$ 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
};

node  *p = new node;
```

![Diagram of a node with fields num, word, and next]
Introduction to linked lists: inserting a node

```c
node *p;
p = new node;
p->num = 5;
p->word = "Ali";
p->next = NULL
```
How can you add another node that is pointed by \texttt{p->link}?

```c
node *p;
p = new node;
p->num = 5;
p->word = "Ali";
p->next = NULL;
node *q;
```
Introduction to linked lists

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

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

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

q = new node;
q->num = 8;
q->word = "Veli";
```
Introduction to linked lists

```c
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

• \( p = \text{new node} \); delete \( p \);
• \( p = \text{new node}[20] \);

• \( p = \text{malloc}(\text{sizeof}(\text{node} )) \); free \( p \);

• \( p = \text{malloc}(\text{sizeof}(\text{node} ) \times 20) \);
• \((p+10)\rightarrow \text{next} = \text{NULL} \); /* 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?
I want to test and understand...

- 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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>/</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prev</td>
<td>5</td>
<td>/</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

head=3

head

8 4 7
Maintaining list of free objects

head=3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>next</strong></td>
<td>/</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>key</strong></td>
<td>7</td>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>prev</strong></td>
<td>5</td>
<td>/</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

free = 6

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>next</strong></td>
<td>/</td>
<td>4</td>
<td>5</td>
<td>/</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td><strong>key</strong></td>
<td>7</td>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>prev</strong></td>
<td>5</td>
<td>/</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

head = 3, free = 6

allocate object:

```
new = free;
free = next[free] ;
```
Multiple lists, single free list

head1=3 => 8, 4, 7
head2=6 => 3, 9
free =2 (2)
Hack: allocate more arrays ...

use integer division and mod

\[ AA[\,(i-1)/7\,] \rightarrow [\,(i-1) \mod 7\,] \]

\[ \text{LIST}(10) = AA[\,1\,][\,2\,] \]
XOR linked lists are a data structure used in computer programming. They take advantage of the bitwise exclusive disjunction (XOR) operation, here denoted by $\oplus$, 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:

```
... A B C D E ...
  -> next  -> next  -> next  ->
  <= prev  <= prev  <= prev  <=
```

An XOR linked list compresses the same information into one address field by storing the bitwise XOR of the address for previous and the address for next in one field:

```
... A B C D E ...
  <= A\oplus C  <= B\oplus D  <= C\oplus E  <=
```

When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field ($B\oplus D$). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.
Queue
(basic idea, does not contain all controls!)

First = List[F]
Pop_first : { return List[F++] }

Last = List[L-1]
Pop_last : { return List[--L] }

Full:  return ( L==MAX_SIZE )
Empty:  F< 0  or   F >= L
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
3   6   |   |   |   |   | 7   5   2
```

First = List[F]

```Add_to_end( x ) : { List[L]=x ; L= (L+1) % MAX_SIZE ] }  // % = modulo```

Last = List[ (L-1+MAX_SIZE) % MAX_SIZE ]

Full: return ( (L+1)%MAX_SIZE == F )

Empty: F==L

```
L, F
|   |   |   |   |   |   | MAX_SIZE-1
```
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

• $5 \ 4 \ 3 \ * \ 2 \ - \ +$ \ \Rightarrow \ \ 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?

3 6 7 5

3 6 7 5 2

• 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 \( n=32 \rightarrow 33 \) (copy 32, insert 1)
• delete: \( 33 \rightarrow 32 \)
  – 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
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= first(D); while( d<=k ) d=next(D);
  – on average \( \leq n/2 \) 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**\( (A[0..N-1], \text{value}, \text{low}, \text{high}) \)

\[
\{
\text{if} \ (\text{high} < \text{low})
\quad \text{return} \ -1 \quad \text{// not found}
\text{mid} = \text{low} + ((\text{high} - \text{low}) / 2) \quad \text{// Note: not (low + high) / 2 !!}
\text{if} \ (A[\text{mid}] > \text{value})
\quad \text{return} \ \text{BinarySearch}(A, \text{value}, \text{low}, \text{mid}-1)
\text{else if} \ (A[\text{mid}] < \text{value})
\quad \text{return} \ \text{BinarySearch}(A, \text{value}, \text{mid}+1, \text{high})
\text{else}
\quad \text{return} \ \text{mid} \quad \text{// found}
\}
\]
Binary search – Iterative

```cpp
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

- \( x \leq A[18] \)
- \( x \leq A[9] \)
- \( x \leq A[13] \)
- \( O(\log n) \)
Sorting

• given a list, arrange values so that
  \[ L[1] \leq L[2] \leq \ldots \leq L[n] \]
• n elements => \( n! \) possible orderings
• One test \( L[i] \leq L[j] \) can divide \( n! \) to 2
  – Make a binary tree and calculate the depth
• \( \log(n!) = O(n \log n) \)
• Hence, lower bound for sorting is \( O(n \log n) \)
  – using comparisons...
  – (proved in previous lecture on blackboard)
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:

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

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

$h = \lg n$

\[
\begin{align*}
T(n) &= cn \\
&= cn/2 + cn/2 \\
&= \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \cdots \\
&= \Theta(n) \\
\end{align*}
\]

Total $= \Theta(n \lg n)$
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 $\leq x \leq$ elements in upper subarray.

   \[
   \begin{array}{ccc}
   \leq x & & \geq x \\
   \end{array}
   \]

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

3. **Combine:** Trivial.

**Key:** Linear-time partitioning subroutine.
Pseudocode for quicksort

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

\[ \text{Partition}(A, p, q) \rightarrow A[p..q] \]

1. \[ x \leftarrow A[p] \]  \( \triangleright \) pivot = \( A[p] \)
2. \[ i \leftarrow p \]
3. for \( j \leftarrow p + 1 \) to \( q \)
   do if \( A[j] \leq x \)
   then \[ i \leftarrow i + 1 \]
   exchange \( A[i] \leftrightarrow A[j] \)
4. exchange \( A[p] \leftrightarrow A[i] \)
5. return \( i \)

**Invariant:**

\[ \begin{array}{|c|c|c|c|c|}
\hline x & \leq x & \geq x & ? \\
\hline p & i & j & q \\
\hline \end{array} \]

Running time

= \( O(n) \) for \( n \) elements.
pivot = A[R];    //
i=L; j=R-1;
while( i<j )
    while ( A[i] <= pivot ) i++ ;
    while ( A[j] > pivot ) j-- ;
A[R]=A[i];
A[R]=pivot;
return i;
Quicksort

- See [http://www.ece.uwaterloo.ca/~ece250/Lectures/Slides/7.06.QuickSort.ppt](http://www.ece.uwaterloo.ca/~ece250/Lectures/Slides/7.06.QuickSort.ppt) for detailed example
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.

\[
T(n) = T(0) + T(n-1) + \Theta(n) \\
= \Theta(1) + T(n-1) + \Theta(n) \\
= T(n-1) + \Theta(n) \\
= \Theta(n^2) \quad \text{(arithmetic series)}
\]
Best-case analysis
(For intuition only!)

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

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

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

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

What is the solution to this recurrence?
Analysis of “almost-best” case

\[ cn \]

\[ T\left(\frac{1}{10} n\right) \quad \text{and} \quad T\left(\frac{9}{10} n\right) \]
Analysis of “almost-best” case

\[ cn \leq T(n) \leq cn \log_{10/9} n + O(n) \]

\[ \Theta(n \log n) \quad \text{Lucky!} \]

September 21, 2005

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson
More intuition

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

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

Solving:

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

How can we make sure we are usually lucky?
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.
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

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$$E[X_k] = \Pr\{X_k = 1\} = 1/n,$$ since all splits are equally likely, assuming elements are distinct.
Analysis (continued)

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

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

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

Take expectations of both sides.
Calculating expectation

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

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

Linearity of expectation.
Calculating expectation

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

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

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

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

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

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

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

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

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

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

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

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

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

\[ = \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n) \quad \text{Summations have identical terms.} \]
Hairy recurrence

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

(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:** \[ \sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \] (exercise).
Substitution method

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

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

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

\[ \leq an \log n , \]

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

- lists – a versatile data structure for various purposes

- Sorting – a typical algorithm (many ways)

- But most of the important (e.g. update) tasks seem to be $O(n)$, which is bad
Can we search faster in linked lists?

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

• Or can it?
A skip list, introduced by Pugh [Pugh 1990], 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 in 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 $\frac{2}{1-p}$ pointers per node. The lists at the higher level act as “express lanes” that allow the sequence of nodes to be traversed quickly. Searching for a node with a particular key involves searching 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 $\frac{1}{1-p}$ nodes are searched on average per level, giving an average search time of $O\left(\log n \frac{1}{(1-p) \log \frac{1}{p}}\right)$ with $n$ nodes at level 0. Skip lists have been extensively studied [Pugh 1990; Papadakis et al. 1990; Devroye 1992; Kirschchenhofer and Prodinger 1994; Kirschchenhofer et al. 1995], and because they require no global balancing operations are particularly useful in parallel systems [Gabarró et al. 1996; Gabarró and Meseguer 1997].

\begin{figure}
\centering
\includegraphics[width=\textwidth]{skip_list}
\caption{A skip list with $n = 6$ nodes and $\lfloor \log n \rfloor = 3$ levels.}
\end{figure}
Skip lists

- Build several lists at different “skip” steps

- $O(n)$ list
- Level 1: $\sim n/2$
- Level 2: $\sim n/4$
- ...
- Level $\log n \sim 2$-$3$ elements...
typedef struct nodeStructure *node;
typedef struct nodeStructure{
    keyType key;
    valueType value;
    node forward[1]; /* variable sized array of forward pointers */
};
Skip Lists

\[ S_0: -\infty \rightarrow 10 \rightarrow 15 \rightarrow 23 \rightarrow 36 \rightarrow +\infty \]
\[ S_1: -\infty \rightarrow 15 \rightarrow 23 \rightarrow +\infty \]
\[ S_2: -\infty \rightarrow 15 \rightarrow +\infty \]
\[ S_3: -\infty \rightarrow +\infty \]
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
What is a Skip List

A skip list for a set $S$ of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that

- Each list $S_i$ contains the special keys $+\infty$ and $-\infty$
- List $S_0$ contains the keys of $S$ in nondecreasing order
- Each list is a subsequence of the previous one, i.e.,
  
  \[ S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h \]

- List $S_h$ contains only the two special keys

We show how to use a skip list to implement the dictionary ADT
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 $y \leftarrow key(after(p))$
  - $x = y$: we return $\text{element}(after(p))$
  - $x > y$: we “scan forward”
  - $x < y$: we “drop down”
- If we try to drop down past the bottom list, we return $\text{NO_SUCH_KEY}$

Example: search for 78
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.
Insertion

To insert an item \((x, o)\) 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 \(i \geq h\), we add to the skip list new lists \(S_{h+1}, \ldots, S_{i+1}\), 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 \(S_0, S_1, \ldots, S_i\).
- For \(j \leftarrow 0, \ldots, i\), we insert item \((x, o)\) into list \(S_j\) after position \(p_j\).

**Example:** insert key 15, with \(i = 2\)
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 $p_j$ is in list $S_j$.
- 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

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 after

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:

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

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

2/17/2009 12:33 AM Skip Lists 94
Height

- The running time of the search and insertion algorithms 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 $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most
  $$n/2^{3\log n} = n/n^3 = 1/n^2.$$ 
- Thus a skip list with $n$ items has height at most $3\log n$ with probability at least $1 - 1/n^2$. 
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.
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.
Skip graphs


Abstract

Skip graphs are a novel distributed data structure, based on skip lists, that provide the full functionality of a balanced tree in a distributed system where resources are stored in separate nodes that may fail at any time. They are designed for use in searching peer-to-peer systems, and by providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash table functionality. Unlike skip lists or other tree data structures, skip graphs are highly resilient, tolerating a large fraction of failed nodes without losing connectivity. In addition, constructing, inserting new nodes into, searching a skip graph, and detecting and repairing errors in the data structure introduced by node failures can be done using simple and straightforward algorithms.

- SODA 2003 proceedings version: [PS](#), [PDF](#).
- Journal version: [PDF](#).

BibTeX

```bibtex
@article{AspnesS2007,
  title="Skip graphs",
  author="James Aspnes and Gauri Shah",
  journal="ACM Transactions on Algorithms",
  month=nov,
  year=2007,
  volume=3,
  number=4,
  pages={(37)}
}
```

Consolidated BibTeX file

Return to James Aspnes's publications

Return to James Aspnes's home page
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