Advanced Algorithmics (6EAP)

Linear structures, sorting, searching, etc

Jaak Vilo
2010 Spring

Lists: Array

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

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)
- Priority queue (fetch highest-value object)

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

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

Lists: Array

- 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

head tail

Singly linked

head tail

Doubly linked

Linked lists: add/delete

size

Operations

• Array indexed from 0 to n – 1:

<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(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</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></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(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(n) )</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:

Doubly-linked list:

  – improvements at operations requiring access to the previous node
  – increases memory requirements...

Improving Efficiency

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

<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(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>delete the ( k )th element</td>
<td>( O(n) )</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
Introduction to linked lists:

- Consider the following struct definition

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

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

Introduction to linked lists: inserting a node

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

Introduction to linked lists: adding a new node

• How can you add another node that is pointed by p->link?

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

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

- \( p = \text{new node} ; \text{delete} \ p ; \)
- \( p = \text{new node}[20] ; \)
- \( p = \text{malloc} (\text{sizeof}(\text{node})) ; \text{free} \ p ; \)
- \( p = \text{malloc} (\text{sizeof}(\text{node}) \times 20) ; \)
- \( (p+10) \rightarrow \text{next} = \text{NULL} ; /* 11\text{th elements} */ \)

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 = \text{new object\_type} ;
- Equals to creating a new object with necessary size of allocated memory (delete can free it)

Some links


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
Maintaining list of free objects

Multiple lists, single free list

Hack: allocate more arrays ...

Queue
(basic idea, does not contain all controls!)

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>6</td>
<td>7</td>
</tr>
</tbody>
</table>

First = List[F]
Add_to_end(x) : { List[L=x ; L= (L+1) % MAX_SIZE ] } \( \% = \text{modulus} \)
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 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) individul 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: \( d=\text{first}(D); \) 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
\[
\begin{array}{l}
\text{BinarySearch}(A[0..N-1], \text{value}, \text{low}, \text{high}) \\
\{ \\
\quad \text{if} (\text{high} < \text{low}) \\
\quad \quad \text{return} -1 // \text{not found} \\
\quad \text{mid} = \text{low} + (\text{high} - \text{low}) / 2) \quad // \text{Note: not (low + high) / 2 !!!} \\
\quad \text{if} (A[mid] > \text{value}) \\
\quad \quad \text{return} \text{BinarySearch}(A, \text{value}, \text{low}, \text{mid}-1) \\
\quad \text{else if} (A[mid] < \text{value}) \\
\quad \quad \text{return} \text{BinarySearch}(A, \text{value}, \text{mid}+1, \text{high}) \\
\quad \text{else} \\
\quad \quad \text{return} \text{mid} // \text{found} \\
\} \\
\end{array}
\]

Binary search – Iterative
\[
\begin{array}{l}
\text{BinarySearch}(A[0..N-1], \text{value}) \\
\{ \\
\quad \text{low} = 0; \ \text{high} = N - 1; \\
\quad \text{while} (\text{low} <= \text{high}) \\
\quad \quad \text{mid} = \text{low} + (\text{high} - \text{low}) / 2) \quad // \text{Note: not (low + high) / 2 !!!} \\
\quad \text{if} (A[mid] > \text{value}) \\
\quad \quad \text{high} = \text{mid} - 1 \\
\quad \text{else if} (A[mid] < \text{value}) \\
\quad \quad \text{low} = \text{mid} + 1 \\
\quad \text{else} \\
\quad \quad \text{return} \text{mid} // \text{found} \\
\} \\
\text{return} -1 // \text{not found} \\
\end{array}
\]

Work performed
• \( x <=> A[18] \) ? <
• \( x <=> A[9] \) ? >
• \( x <=> A[13] \) ? ==

\[
\begin{array}{c}
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \\
\end{array}
\]
\[
\begin{array}{c}
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 19 \\
\end{array}
\]
\[
\begin{array}{c}
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 13 \\
\end{array}
\]

• \( O(\log n) \)
Sorting

- given a list, arrange values so that
  \( L[1] \leq L[2] \leq ... \leq L[n] \)
- \( n \) 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
- \( \log(n!) = \Theta(n \log n) \)
- Hence, lower bound for sorting is \( \Theta(n \log n) \)
  - using comparisons...
  - (proved in previous lecture on blackboard)

Decision tree model

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

- \( \log(n!) = \log(n) + \log(n-1) + ... + \log(1) \)
  a) \( \leq n \log(n) \)
  b) \( \geq \frac{n}{2} \log \left( \frac{n}{2} \right) = \frac{n}{2} \log n - \frac{n}{2} \)

Lower bound for decision-tree sorting

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

**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, \( n! \leq 2^h \).

\[ h \geq \log(n!) \geq \log((n/e)^n) \geq n \log(n) - n \log e = \Omega(n \log n) \]
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:

```
A: 1 2 3 4 5 6 7 8
B: 9 10 11 12 13 14 15 16
```

```
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:
  \[
  T(n) = \begin{cases} 
  O(1) & n = 1 \\
  2T(\frac{n}{2}) + O(n) & n > 1 
  \end{cases}
  \]
Recursion tree

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

$T(n) = \Theta(n \log n)$

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/

Quick sort

- 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 $< x$ elements in upper subarray
2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.

Key: Linear-time partitioning subroutine.

Partitioning subroutine

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++ ;  // 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.

\[
T(n) = T(0) + T(n-1) + \Theta(n)
\]

\[
= \Theta(1) + T(n-1) + \Theta(n)
\]

\[
= T(n-1) + \Theta(n)
\]

\[
= \Theta(n^2) \quad \text{(arithmetic series)}
\]

### 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 \( \frac{1}{10} : \frac{9}{10} \)?

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

What is the solution to this recurrence?

### Analysis of “almost-best” case

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

\[
\log_{10}n \quad \Theta(1)
\]

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

\[
cn \leq T(n) \leq cn \log_{10}n + \Theta(n)
\]
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:

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

How can we make sure we are usually lucky?

Choice of pivot

• Select median of three ...

• Select random – opponent 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.

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

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

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

Analysis (continued)

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

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

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

Take expectations of both sides.

Calculating expectation

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

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

Calculating expectation

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

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

Calculating expectation

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

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

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

**Use fact:** \(\sum_{k=2}^{n-1} k \log k \leq 1/2 n^2 \log n - 1/2 n^2\) (exercise).
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:** $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output:** $B[1 \ldots n]$, sorted.
- **Auxiliary storage:** $C[1 \ldots k]$.

Counting sort

```
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

\[
\begin{align*}
A: & \quad 4 \ 1 \ 3 \ 4 \ 3 \\
B: & \quad \text{Empty}
\end{align*}
\]

for \( i \leftarrow 1 \) to \( k \)
\[
\text{do } C[i] \leftarrow 0
\]

Loop 2

\[
\begin{align*}
A: & \quad 4 \ 1 \ 3 \ 4 \ 3 \\
B: & \quad \text{Empty}
\end{align*}
\]

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

Loop 3

\[
\begin{align*}
A: & \quad 4 \ 1 \ 3 \ 4 \ 3 \\
B: & \quad \text{Empty}
\end{align*}
\]

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

Loop 4

\[
\begin{align*}
A: & \quad 4 \ 1 \ 3 \ 4 \ 3 \\
B: & \quad 3 \ 4 \ 4
\end{align*}
\]

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

Analysis

\[
\begin{align*}
\Theta(k) & \quad \{ \text{for } i \leftarrow 1 \text{ to } k \} \\
& \quad \text{do } C[i] \leftarrow 0
\end{align*}
\]

\[
\begin{align*}
\Theta(n) & \quad \{ \text{for } j \leftarrow 1 \text{ to } n \} \\
& \quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1
\end{align*}
\]

\[
\begin{align*}
\Theta(k) & \quad \{ \text{for } i \leftarrow 2 \text{ to } k \} \\
& \quad \text{do } C[i] \leftarrow C[i] + C[i-1]
\end{align*}
\]

\[
\begin{align*}
\Theta(n) & \quad \{ \text{for } j \leftarrow n \text{ downto } 1 \} \\
& \quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
& \quad C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}
\]

\[
\Theta(n+k)
\]

Running time

If \( k = O(n) \), then counting sort takes \( \Theta(n) \) time.

- But, sorting takes \( \Omega(n \ lg \ n) \) time!
- Where’s the fallacy?

Answer:

- \textit{Comparison sorting} takes \( \Omega(n \ lg \ n) \) time.
- Counting sort is not a \textit{comparison sort}.
- In fact, not a single comparison between elements occurs!
Radix sort

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

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.

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.
  - Two numbers equal in digit \( t \) are put in the same order as the input ⇒ correct order.
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

| 8 | 8 | 8 | 8 |

$r = 8 \Rightarrow b/r = 4$ 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 $r$

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

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

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ 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 $\lceil \lg 2000 \rceil = 11$ 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. \( a \)
2. \( b \)
3. \( c \)
4. \( d \)

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 bitwissort(SORTTYPE *ARRAY, int size)
{
    int i, 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 */
    for each bit */
    i = 0;
    new_mask:
    for (i = 0; i < size; i++) { /* Advance while bit == 0 */
        if (ARRAY[i] & curbit) { /* Save current prefix snapshot */
            array_end = i;
            /* End of first pass */
            if (i == array_end) { /* Realend of array */
                if (ARRAY[i] & mask) { /* Group goes new_mask */
                    /* new prefix */
                    if (ARRAY[i] & curbit) { /* Bit i is - need to swap with previous location of 1, A[j] = A[i] */
                        tmp = ARRAY[i];
                        ARRAY[i] = ARRAY[j];
                        ARRAY[j] = tmp;
                        i++;
                    } else { /* Swap and increase i */
                        i++;
                    }
                }
            } else { /* Swap and increase i */
                tmp = ARRAY[i];
                ARRAY[i] = ARRAY[i-1];
                ARRAY[i-1] = tmp;
            }
        }
        mask = mask & curbit; /* area under mask is new sorted */
        curbit = curbit >> 1; /* next bit */
    }
}

Bitwise from left to right

0010000
0010001
0101000
0101001
1001000
1001001
1001010
1001011
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

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

Sort Benchmark

- http://sortbenchmark.org/
  - Sort Benchmark Home Page
  - We have a new benchmark called new GraySort, new 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
  - 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.
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 – $A[\lfloor (n+1)/2 \rfloor]$ or $A\lfloor (n+1)/2 \rfloor$

Min and max together

- compare every two elements $A[i], A[i+1]$
- Compare larger against current max
- Smaller against current min
- $3n/2$

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 $i \leq k$
  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
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 sequence 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 double-linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $\phi_p$ pointers per node. The lists at the higher level are in “reverse order” 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 $\phi_p$ nodes are searched on average per level, giving an average search time of $O\left(\frac{1}{1-p}\right)$ with a node at level 0. Skip lists have been extensively analyzed (Pagh 1999; Pagh et al. 1996; Demaine 1997; Kienzle and Friediger 1994; Brodnik et al. 1995), and because they require no global synchronizing operations are particularly useful in parallel systems (Karakus et al. 1996; Guha and Miserendino 1997).

![Skip List Diagram](image)

Skip List

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

Skip Lists

![Skip Lists](image)
Outline and Reading

- What is a skip list ($\S\ 3.5$)
- Operations
  - Search ($\S\ 3.5.1$)
  - Insertion ($\S\ 3.5.2$)
  - Deletion ($\S\ 3.5.2$)
- Implementation
- Analysis ($\S\ 3.5.3$)
  - 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 $y = \text{keyAfter}(p)$
    - $x = y$ return $\text{elementAfter}(p)$
    - $x < y$ we "scan forward"
    - $x > y$ we "drop down"
  - If we try to drop down past the bottom list, return $\text{NO\_SEARCH\_KEY}$

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 $i > k$, we add to the skip list new lists $S_{p_0}, S_{p_1}, \ldots, S_{p_i}$ 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 $S_{p_0}, S_{p_1}, \ldots, S_{p_i}$
  - For $j = 0, 1, \ldots, i$, 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 if 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 $S_j$
  - 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

Outline and Reading

- What is a skip list ($\S\ 3.5$)
- Operations
  - Search ($\S\ 3.5.1$)
  - Insertion ($\S\ 3.5.2$)
  - Deletion ($\S\ 3.5.2$)
- Implementation
- Analysis ($\S\ 3.5.3$)
  - 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 $y = \text{keyAfter}(p)$
    - $x = y$ return $\text{elementAfter}(p)$
    - $x < y$ we "scan forward"
    - $x > y$ we "drop down"
  - If we try to drop down past the bottom list, return $\text{NO\_SEARCH\_KEY}$

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 $i > k$, we add to the skip list new lists $S_{p_0}, S_{p_1}, \ldots, S_{p_i}$ 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 $S_{p_0}, S_{p_1}, \ldots, S_{p_i}$
  - For $j = 0, 1, \ldots, i$, 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 if 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 $S_j$
  - 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 above
  - link to the node below
- 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 L_i with probability 1/2^i.
  - By Fact 2, the expected size of list L_i is n/2^i.
- The expected number of nodes used by the skip list is

\[
\sum_{i=0}^{\log n} \frac{n}{2^i} = 2n
\]

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

Height

- The running time of the search an 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 (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 L_i with probability 1/2^i.
  - By Fact 3, the probability that list L_i has at least one item is at most n/2^i.
  - By picking i = 3log n, we have that the probability that L_{3log n} has at least one item is at most n/2^{3log n} = 1/2.
  - Thus, a skip list with n items has height at most log n with probability at least 1 - 1/2.

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 p.
- 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.
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