Introduction to Algorithms

Chapter 10: Elementary Data Structures - Review

Introduction

- Data structures for dynamic sets
- Set in mathematics
  - \{1, 2, 5, 4, 3, 6\}
- Set in algorithms
  - Allow repetition in set elements: \{1, 2, 5, 4, 3, 6, 4\}
  - Dynamic: can grow, shrink, or change over time
  - Set operations: insert, delete, test membership

Elements of A Dynamic Set

- Each element is represented by an object (or record)
  - An object may consist of many fields
  - Need a pointer to an object to examine and manipulate its fields
  - Key field for identifying objects and for the set manipulation
    - Keys are usually drawn from a totally ordered set
  - Satellite fields: all the fields irrelevant for the set manipulation
- Type Record patron {
  integer      patron_ID;
  char[20]    name;
  integer age;
  char[10]   department;…
}

Record(Object) and Pointer

Type Record patron {
  integer      patron_ID;
  char[20]    name;
  integer age;
  char[10]   department;…
}

Type patron p1, p2, p3, p4;

Type pointer *p;

<table>
<thead>
<tr>
<th>Patron_ID</th>
<th>Name</th>
<th>Age</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>T8403</td>
<td>Claven</td>
<td>20</td>
<td>Library</td>
</tr>
</tbody>
</table>
Operations on Dynamic Sets

- Query operations: return information about a set
  - SEARCH(S, k): given a set S and key value k, returns a pointer x to an element in S such that key[x] = k, or NIL if no such element belongs to S
  - MINIMUM(S): returns a pointer to the element of S with the smallest key
  - MAXIMUM(S): returns a pointer to the element of S with the largest key
  - SUCCESSOR(S, x): returns a pointer to the next larger element in S, or NIL if x is the maximum element
  - PREDECESSOR(S, x): returns a pointer to the next smaller element in S, or NIL if x is the minimum element

Operations on Dynamic Sets (Cont.)

- Modifying operations: change a set
  - INSERT(S, x): augments the set S with the element pointed to by x. We usually assume that any fields in element x needed by the set implementation have already initialized.
  - DELETE(S, x): given a pointer x to an element in the set S, removes x from S.

Introduction

- Stack
  - The element deleted from the set is the one most recently inserted
  - Last-in, First-out (LIFO)
- Stack operations
  - PUSH: Insert
  - DELETE: Delete
  - TOP: return the key value of the most recently inserted element
  - STACK-EMPTY: check if the stack is empty
  - STACK-FULL: check if the stack is full
Represent Stack by Array

- A stack of at most $n$ elements can be implemented by an array $S[1..n]$
  - $top[S]$: a pointer to the most recently inserted element
  - A stack consists of elements $S[1..top[S]]$

Stack Operations

STACK-EMPTY($S$)
1. if $top[S] = 0$
2. then return TRUE
3. else return FALSE

POP($S$)
1. if STACK-EMPTY($S$)
2. then error "underflow"
3. else $top[S] \leftarrow top[S] - 1$
4. return $S[top[S] + 1]$

How to implement TOP($S$), STACK-FULL($S$) ?

$O(1)$

Illustration of PUSH

PUSH($S$, $x$)
1. $top[S] \leftarrow top[S] + 1$
2. $S[top[S]] \leftarrow x$

Illustration of POP

POP($S$)
1. $top[S] \leftarrow top[S] - 1$ (top[S] = 5)
2. return $S[top[S] + 1]$ (return S[6])
Introduction

- Queue
  - The element deleted is always the one that has been in the set for the longest time
  - First-in, First-out (FIFO)
- Queue operations
  - ENQUEUE: Insert
  - DEQUEUE: Delete
  - HEAD: return the key value of the element that has been in the set for the longest time
  - TAIL: return the key value of the element that has been in the set for the shortest time
  - QUEUE-EMPTY: check if the stack is empty
  - QUEUE-FULL: check if the stack is full

Represent Queue by Array

- A queue of at most n-1 elements can be implemented by an array S[1..n]
  - head[S]: a pointer to the element that has been in the set for the longest time
  - tail[S]: a pointer to the next location at which a newly arriving element will be inserted into the queue
  - The elements in the queue are in locations head[Q], head[Q]+1, ..., tail[Q]-1
  - The array is circular
- Empty queue: head[Q] = tail[Q]
  - Initially we have head[Q] = tail[Q] = 1
- Full queue: head[Q] = tail[Q] + 1 (in circular sense)

Illustration of A Queue

Figure 10.2. A queue implemented using an array Q[1..12]. Queue elements appear only in the lightly shaded positions. (a) The queue has 5 elements, in locations Q[7..11]. (b) The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). (c) The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.
Queue Operations

**ENQUEUE(Q, x)**

1. $Q[tail(Q)] \leftarrow x$
2. if $tail(Q) = length(Q)$
3. \quad then $tail(Q) \leftarrow 1$
4. \quad else $tail(Q) \leftarrow tail(Q) + 1$

**DEQUEUE(Q)**

1. $x \leftarrow Q[head(Q)]$
2. if $head(Q) = length(Q)$
3. \quad then $head(Q) \leftarrow 1$
4. \quad else $head(Q) \leftarrow head(Q) + 1$
5. return $x$

**O(1)**

Linked Lists

Introduction

- A linked list is a data structure in which the objects are arranged in linear order
  - The order in a linked list is determined by pointers in each object
- Doubly linked list
  - Each element is an object with a *key* field and two other pointer fields: *next* and *prev*, among other satellite fields. Given an element $x$
    - *next*[x] points to its successor
    - if $x$ is the last element (called *tail*), *next*[x] = NIL
    - *prev*[x] points to its predecessor
      - if $x$ is the first element (called *head*), *prev*[x] = NIL
- An attribute head[L] points to the first element of the list
  - if head[L] = NIL, the list is empty

Introduction (Cont.)

- Singly linked list: omit the *prev* pointer in each element
- Sorted linked list: the linear order of the list corresponds to the linear order of keys stored in elements of the list
  - The minimum element is the head
  - The maximum element is the tail
- Circular linked list: the *prev* pointer of the head points to the tail, and the *next* pointer of the tail points to the head
Illustration of A Doubly Linked List

(a) head[L] ───> 9 ───> 16 ───> 4 ───> 1

(b) head[L] ───> 25 ───> 9 ───> 16 ───> 4

(c) head[L] ───> 25 ───> 9 ───> 16 ───> 1

Figure 10.3 (a) A doubly linked list L representing the dynamic set {1, 4, 9, 16}. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The next field of the tail and the prev field of the head are NIL, indicated by a diagonal slash. The attribute head[L] points to the head. (b) Following the execution of LIST-INSERT(L, x), where key[x] = 25, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Searching A Linked List

- LIST-SEARCH(L, k): finds the first element with key k in list L by a simple linear search, returning a pointer to this element.
  - If no object with key k appears in the list, then NIL

    LIST-SEARCH(L, k)
    1. x ← head[L]
    2. while x ≠ NIL and key[x] ≠ k
    3.   do x ← next[x]
    4. return x

Inserting Into A Linked List

- LIST-INSERT(L, x): given an element pointed by x, splice x onto the front of the linked list.

    LIST-INSERT(L, x)
    1. next[x] ← head[L]
    2. if head[L] ≠ NIL
    3.   then prev[head[L]] ← x
    4. head[L] ← x
    5. prev[x] ← NIL

How about LIST-SEARCH(L, 7)?
Illustration of LIST-INSERT

LIST-INSERT(L, x)

\[ \text{prev}[x] \gets \text{NIL} \]

Illustration of LIST-DELETE

LIST-DELETE(L, x)

1. if prev[x] ≠ NIL
2. then next[prev[x]] ← next[x]
3. else head[L] ← next[x]
4. if next[x] ≠ NIL
5. then prev[next[x]] ← prev[x]

Implementing Pointers and Objects

Need garbage collection for x
Pointers in Pseudo Language

Type Record patron {  
  integer patron_ID;  
  char[20] name;  
  integer age;  
  char[10] department;  
  ...  
}

Type patron p1, p2, p3, p4;

Type patron *pointer_to_p1

Some languages, like C and C++, support pointers and objects; but some others not

A Multiple-Array Representation of Objects

- We can represent a collection of objects that have the same fields by using an array for each field.
  - Figure 10.3 (a) and Figure 10.5
    - For a given array index x, key[x], next[x], and prev[x] represent an object in the linked list
    - A pointer x is simply a common index on the key, next, and prev arrays
    - NIL can be represented by an integer that cannot possibly represent an actual index into the array

A Multiple-Array Representation of Objects Example

- An object occupies a contiguous set of locations in a single array \( \mathbf{A[j..k]} \)
  - A pointer is simply the address of the first memory location of the object \( \mathbf{A[j]} \)
  - Other memory locations within the object can be indexed by adding an offset to the pointer \( 0 \sim k-j \)
  - Flexible but more difficult to manage

A Single-Array Representation of Objects
Allocating and Freeing Objects

- To insert a key into a dynamic set represented by a linked list, we must allocate a pointer to a currently unused object in the linked-list representation.
  - It is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated.
- Allocate and free homogeneous objects using the example of a doubly linked list represented by multiple arrays.
  - The arrays in the multiple-array representation have length \( m \).
  - At some moment the dynamic set contains \( n \leq m \) elements.
  - The remaining \( m - n \) objects are free, can be used to represent elements inserted into the dynamic set in the future.

Free List

- A singly linked list to keep the free objects.
  - Initially it contains all \( n \) unallocated objects.
- The free list is a stack.
  - Allocate an object from the free list \( \Rightarrow \; \text{POP} \).
  - De-allocate (free) an object \( \Rightarrow \; \text{PUSH} \).
  - The next object allocated is the last one freed.
- Use only the next array to implement the free list.
- A variable free pointers to the first element in the free list.
- Each object is either in list \( L \) or in the free list, but not in both.

Free List Example

Allocate And Free An Object

```
Allocate-Object()
1    if free = NIL
2    then error “out of space”
3    else x ← free
4    free ← next[x]
5    return x

Free-Object(x)
1    next[x] ← free
2    free ← x
```
Two Linked Lists $L_1$ and $L_2$, and A Free List

```
<table>
<thead>
<tr>
<th>free</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_2</td>
<td>9</td>
</tr>
<tr>
<td>key</td>
<td>k3 k2 k1</td>
</tr>
<tr>
<td>prev</td>
<td>7 7 7</td>
</tr>
<tr>
<td>L_1</td>
<td>3</td>
</tr>
<tr>
<td>key</td>
<td>k3</td>
</tr>
<tr>
<td>prev</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Representing Rooted Trees

Binary Tree

- Use linked data structures to represent a rooted tree
  - Each node of a tree is represented by an object
  - Each node contains a key field and maybe other satellite fields
  - Each node also contains pointers to other nodes
- For binary tree...
  - Three pointer fields
    - parent: pointer to the parent $\rightarrow$ NIL for root
    - left: pointer to the left child $\rightarrow$ NIL if no left child
    - right: pointer to the right child $\rightarrow$ NIL if no right child
  - root[T] pointer to the root of the tree
    - NIL for empty tree

Figure 10.9 The representation of a binary tree $T$. Each node $x$ has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The key fields are not shown.
**Draw the Binary Tree Rooted At Index 6**

<table>
<thead>
<tr>
<th>Index</th>
<th>Key</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>8</td>
<td>NIL</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>10</td>
<td>NIL</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>NIL</td>
<td>NIL</td>
</tr>
</tbody>
</table>

**Rooted Trees With Unbounded Branches**

- The representation for binary trees can be extended to a tree in which no. of children of each node is at most $k$
  - left, right $\Rightarrow$ child$_1$, child$_2$, ..., child$_k$
- If no. of children of a node can be unbounded, or $k$ is large but most nodes have small numbers of children...
  - Left-child, right sibling representation
    - Three pointer fields
      - $p$: pointer to the parent
      - left-child: pointer to the leftmost child
      - right-sibling: pointer to the sibling immediately to the right
    - root[T] pointer to the root of the tree
    - $O(N)$ space for any $n$-node rooted tree

---

*Figure 10.10*  The left-child, right-sibling representation of a tree $T$. Each node $x$ has fields $p[x]$ (top), $\text{left-child}[x]$ (lower left), and $\text{right-sibling}[x]$ (lower right). Keys are not shown.