Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

Stacks and Queues

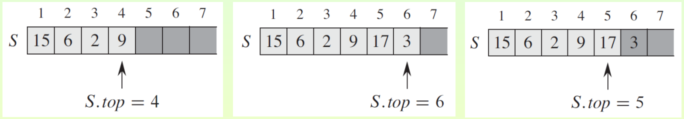
Stacks and LIFO versus Queues and FIFO

- Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified.
  - In a **stack**, the element deleted from the set is the one most recently inserted:
    The stack implements a **last-in, first-out**, or **LIFO**, policy.
  - In a **queue**, the element deleted is always the one that has been in the set for the longest time:
    The queue implements a **first-in, first-out**, or **FIFO**, policy.
- There are several efficient ways to implement stacks and queues on a computer:

  We show how to use a simple array to implement each.

Stacks

- The INSERT operation on a stack is often called PUSH, and the DELETE operation, without an argument, is often called POP.
- We can implement a stack of at most n elements with an array $S[1 \ldots n]$.



- The array has an attribute S.top that indexes the most recently inserted element.
- The stack consists of elements S[1 . . . S.top], where:
  - S[1] is the element at the bottom of the stack;
  - S[S.top] is the element at the top.
- When S.top = 0, the stack contains no elements and is **empty**.

Stacks: Test and Operations

- To test whether the stack is empty we use:

```
STACKEMPTY(S)
1. if S.top == 0
2.     return TRUE
3. else return FALSE
```

- If we attempt to pop an empty stack, the stack **underflows**.

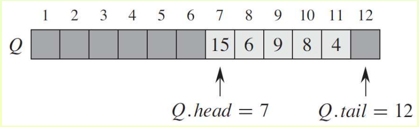
```
POP(S)
1. if STACKEMPTY(S)
2.     error "underflow"
3. else S.top = S.top − 1
4.     return S[S.top + 1]
```

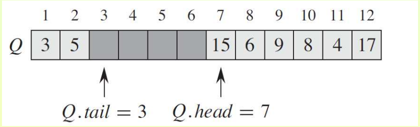
- If S.top exceeds n, the stack **overflows** (we ignore this).

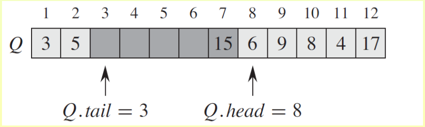
```
PUSH(S, $x$)
1. S.top = S.top + 1
2. S[S.top] = $x$
```

Queues

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation, that takes no argument, DEQUEUE.
- The queue has a **head** and a **tail**.
- When an element is enqueued, it enters at the tail of the queue.
- The element dequeued is always the one at the head of the queue.
- One way to implement a queue of at most $n-1$ elements is using an array Q$[1 \ldots n]$.







Implementing Queues

- The queue has an attribute Q.head that indexes its head.
- The attribute Q.tail indexes the next location at which a newly arriving element will be inserted into the queue.
- The elements in the queue reside in locations

$$\text{Q.head}, \text{Q.head} + 1, \ldots, \text{Q.tail} - 1,$$

  where we "wrap around" in the sense that location 1 immediately follows location n in a circular order.
- When Q.head = Q.tail, the queue is empty.
- Initially, we have Q.head = Q.tail = 1.
- If we attempt to dequeue an element from an empty queue, the queue **underflows**.
- When Q.head = Q.tail + 1, the queue is full, and if we attempt to enqueue an element, then the queue **overflows**.

Operations on Queues

- In our procedures EnQueue and DeQueue, we have
  - assumed that $n = Q.\text{length}$;
  - omitted the error checking for underflow and overflow.

EnQueue(Q, x)

```
1. $Q[Q.\text{tail}] = x$
2. if $Q.\text{tail} == Q.\text{length}$
3.     $Q.\text{tail} = 1$
4. else $Q.\text{tail} = Q.\text{tail} + 1$
```

DeQueue(Q)

```
1. $x = Q[Q.\text{head}]$
2. if $Q.\text{head} == Q.\text{length}$
3.     $Q.\text{head} = 1$
4. else $Q.\text{head} = Q.\text{head} + 1$
5. return $x$
```

- Each operation takes $O(1)$ time.

Subsection 2

Linked Lists

Linked Lists

- A **linked list** is a data structure in which the objects are arranged in a linear order.
- The linear order is not determined by indices as in an array, but by a pointer in each object.
- Each element of a **doubly linked list** L is an object with an attribute key and two pointer attributes next and prev.
- Given an element x in the list, x.next points to its successor in the linked list, and x.prev points to its predecessor.
- If x.prev = NIL, the element x has no predecessor and is therefore the first element, or **head**, of the list.
- If x.next = NIL, the element x has no successor and is therefore the last element, or **tail**, of the list.
- An attribute L.head points to the first element of the list. If L.head = NIL, the list is empty.

Types of Linked Lists

- A list may have one of several forms:
    - It may be either **singly linked** or **doubly linked**.
        - If a list is singly linked, we omit the prev pointer in each element.
    - It may be **sorted** or not.
        - If a list is sorted, the linear order of the list corresponds to the linear order of keys stored in elements of the list.
          The minimum element is then the head of the list, and the maximum element is the tail.
        - If the list is unsorted, the elements can appear in any order.
    - It may be **circular** or not.
        - In a circular list, the prev pointer of the head of the list points to the tail, and the next pointer of the tail of the list points to the head.
- We will be working with unsorted, doubly linked lists.

Searching a Doubly Linked List

- The procedure ListSearch(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 is returned.

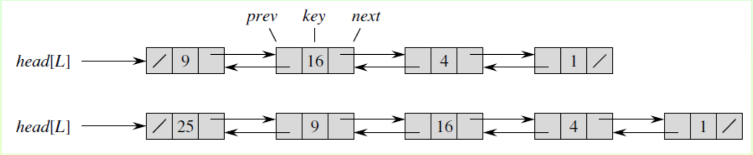
ListSearch(L, k)

```
1. $x$ = L.head
2. while $x \neq$ NIL and $x$.key $\neq k$
3.     $x$ = $x$.next
4. return $x$
```

- To search a list of n objects, the ListSearch procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Inserting in a Doubly Linked List

- Given an element x whose key field has already been set, the ListInsert procedure "splices" x onto the front of the linked list:



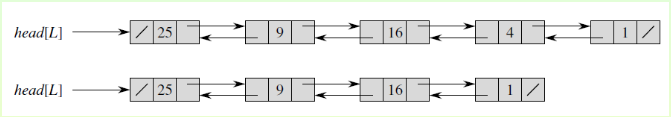
ListInsert(L, x)

```
1. $x$.next = L.head
2. if L.head $\neq$ NIL
3.     L.head.prev = $x$
4. L.head = $x$
5. $x$.prev = NIL
```

- The running time for ListInsert on a list of n elements is $O(1)$.

Deleting from a Linked List

- ListDelete removes an element x from a linked list L. Given a pointer to x, it "splices" x out of the list by updating pointers.
- If we wish to delete an element with a given key, we must first call ListSearch to retrieve a pointer to the element.



ListDelete(L, x)

```
1. if $x$.prev $\neq$ NIL
2.     $x$.prev.next $=$ $x$.next
3. else L.head $=$ $x$.next
4. if $x$.next $\neq$ NIL
5.     $x$.next.prev $=$ $x$.prev
```

- ListDelete runs in $O(1)$ time. However, deletion by key takes $\Theta(n)$ worst case time due to first calling ListSearch.

Sentinels

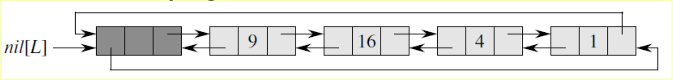
- The code for LISTDELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

```
LISTDELETE′(L, x)
1. x.prev.next = x.next
2. x.next.prev = x.prev
```

- A **sentinel** is a dummy object that allows us to simplify boundary conditions:

  Suppose that we provide with list L an object L.nil that represents NIL but has all the attributes of the other objects in the list.
  Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel L.nil.

- This change turns a regular doubly linked list into a circular doubly linked list with L.nil lying between head and tail.



Searching and Inserting with Sentinels

- The code for ListSearch remains the same, but with the references to NIL and L.head changed as specified above:

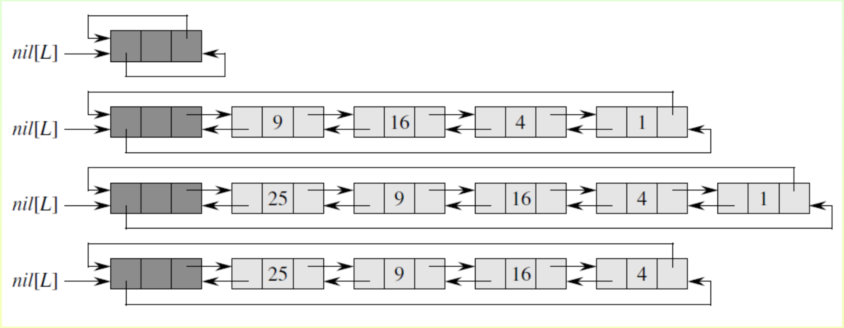
```
ListSearch′(L, $k$)
1. $x$ = L.nil.next
2. while $x \neq$ L.nil and $x$.key $\neq k$
3.     $x = x$.next
4. return $x$
```

- We use the two-line procedure ListDelete′ from before to delete an element from the list.
- For insertion:

```
ListInsert′(L, $x$)
1. $x$.next = L.nil.next
2. L.nil.next.prev = $x$
3. L.nil.next = $x$
4. $x$.prev = L.nil
```

Example of Inserting and Deleting with Sentinels

- The figure shows the effects of LISTINSERT′ and LISTDELETE′ on a sample list:



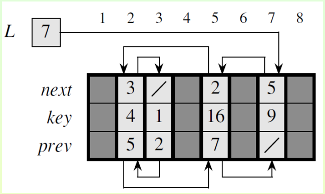
Subsection 3

Implementing Pointers and Objects

A Multiple-Array Representation of Objects

- We can represent a collection of objects that have the same attributes by using an array for each attribute.

  The figure shows how we can implement a linked list with three arrays. The array key holds the values of the keys currently in the dynamic set, and the pointers reside in the arrays next and prev.

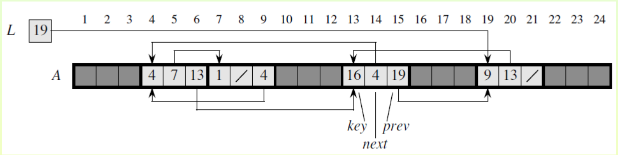
  For a given array index x, the array entries x.key, x.next and x.prev represent an object in the linked list.

  For the constant NIL, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays.

  A variable L holds the index of the head of the list.

A Single-Array Representation of Objects

- The words in a computer memory are typically addressed by integers from 0 to $M - 1$, where M is a suitably large integer.
- An object may occupy a contiguous set of locations in memory.
  - A pointer is simply the address of the first memory location.
  - Other locations can be indexed by adding an offset to the pointer.
- The figure shows how a single array A can be used to store the linked list:

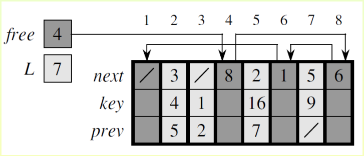


  An object occupies a contiguous subarray A[$j \dots k$]. Each field of the object corresponds to an offset in the range from 0 to $k - j$, and a pointer to the object is the index j.
- The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array.

Allocating and Freeing Objects

- We explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.
- Suppose that the arrays in the multiple array representation have length m and that at some moment the dynamic set contains $n \leq m$ elements. Then n objects represent elements currently in the dynamic set, and the remaining $m - n$ objects are **free**. The free objects are kept in a singly linked list, the **free list**.

  The free list uses only the next array, which stores the next pointers within the list. The head of the free list is held in the global variable free.



- When the dynamic set represented by linked list L is nonempty, the free list may be intertwined with list L.

Allocate and Free Procedures

- The free list is a stack: The next object allocated is the last one freed.

  We can use a list implementation of the stack operations PUSH and POP to implement the procedures for allocating and freeing objects.
- The global variable free used in the following procedures points to the first element of the free list.

ALLOCATEOBJECT()

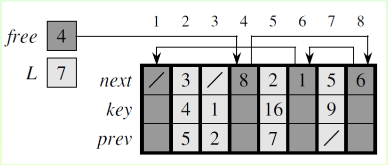
```
1. if free = NIL
2.    error "out of space"
3. else $x$ = free
4.    free = $x$.next
5.    return $x$
```

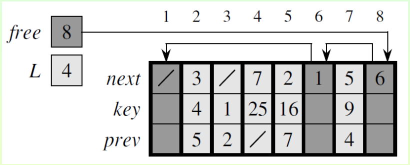
FREEOBJECT(x)

```
1. $x$.next = free
2. free = $x$
```

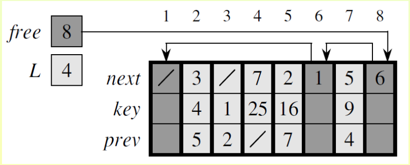
Illustration of Allocating and Freeing

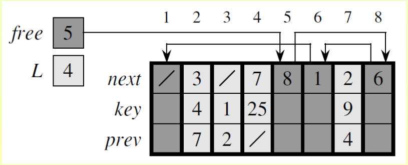
- Calling ALLOCATEOBJECT() (returns index 4), setting key[4] to 25, and calling LISTINSERT(L, 4).





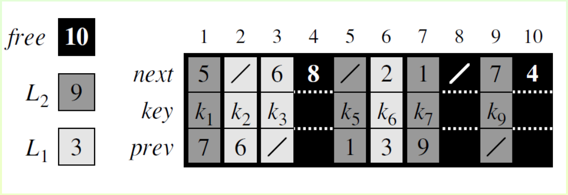
- After executing LISTDELETE(L, 5), we call FREEOBJECT(5), making object 5 the new free-list head, with object 8 following it:





A Free List Serving Multiple Linked Lists

- It is common to use a single free list to service several linked lists.
- The figure shows two linked lists and a free list intertwined through key, next and prev arrays.



- The two procedures $\textsc{AllocateObject}$ and $\textsc{FreeObject}$ run in $O(1)$ time.
- They can be modified to work for any homogeneous collection of objects by letting any one of the fields in the object act like a next field in the free list.

Subsection 4

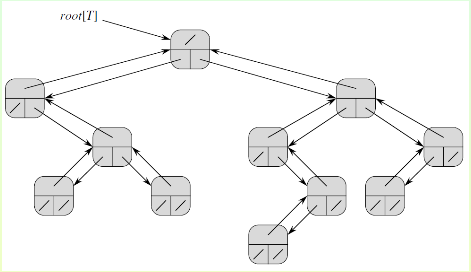
Representing Rooted Trees

Representing Trees by Linked Data Structures

- We look at the problem of representing rooted trees by linked data structures:
  - We first look at binary trees.
  - Then we present a method for rooted trees in which nodes can have an arbitrary number of children.
- We represent each node of a tree by an object.
- As with linked lists, we assume that:
  - Each node contains a key attribute.
  - The remaining attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary Trees

- The figure shows how we use the attributes p, left and right to store pointers to the parent, left child, and right child of each node in a binary tree T:



- If x.p $=$ NIL, then x is the root.
- If node x has no left child, then x.left $=$ NIL, and similarly for the right child.
- The root of the entire tree T is pointed to by the attribute T.root.

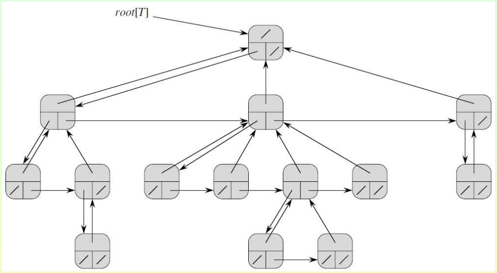
  If T.root $=$ NIL, then the tree is empty.

Trees with Unbounded Branching: Inefficient Attempts

- To represent a tree in which the number of children of each node is at most some constant k, we replace the left and right attributes by child_1, child_2, …, child_k.
- This scheme no longer works when the number of children of a node is unbounded, since we do not know how many attributes to allocate in advance.
- Moreover, even if the number of children k is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.
- A scheme to represent trees with arbitrary numbers of children that uses only $O(n)$ space for any n-node rooted tree is given in the following slide.

Rooted Trees with Unbounded Branching

- The **left-child, right-sibling representation**:



As before, each node contains a parent pointer p, and T.root points to the root of tree T.
Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. x.LeftChild points to the leftmost child of node x.
   If node x has no children, then x.LeftChild = NIL.
2. x.RightSibling points to the sibling of x immediately to its right.
   If node x is the rightmost child of its parent, then x.RightSibling = NIL.