Lecture 7: Data Structures

MCS/CS 401 Computer Algorithms I — June 29, 2022
Instructor: Danko Adrovic

Lecture Outline

- Sets and Fundamental Data Structures
  - Stack
  - Queue
  - Linked Lists
  - Linked List Operations
  - Circular Linked Lists
- Representing rooted trees

1 Sets and Fundamental Data Structures

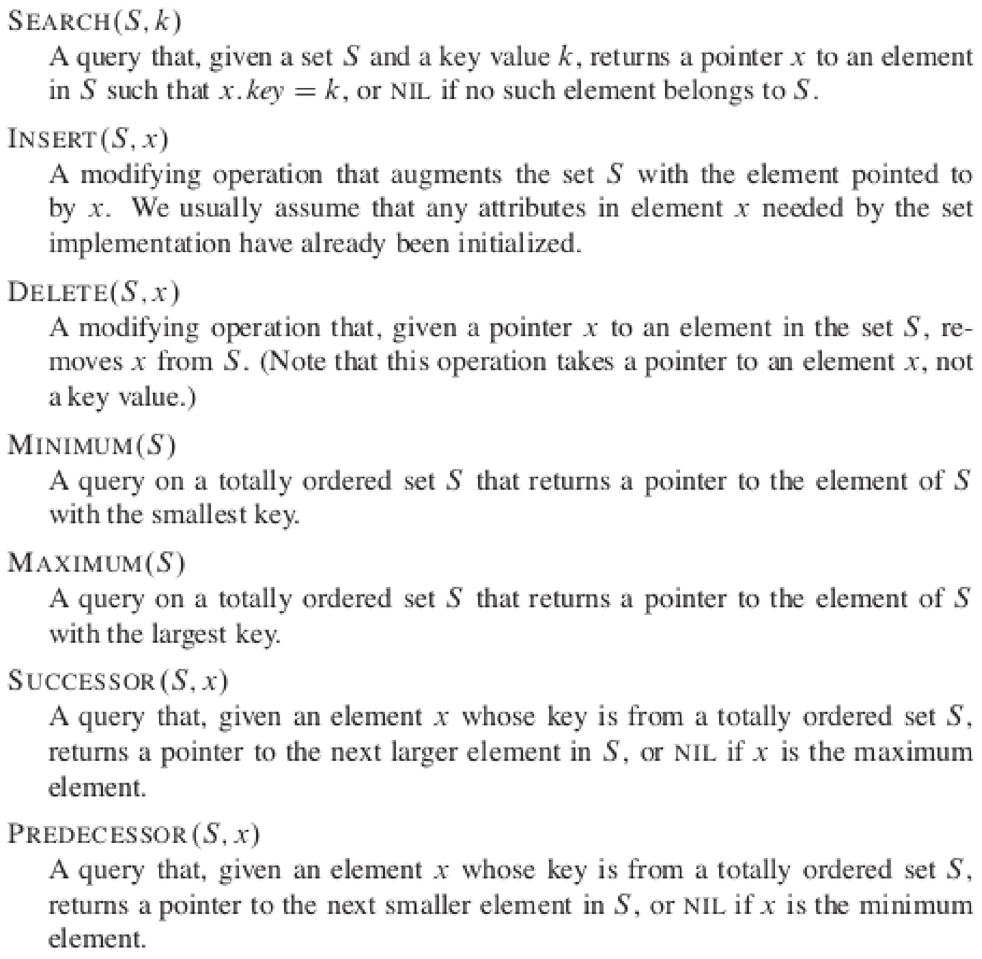
In this lecture we are going to cover Chapter 10, which addresses some of the most fundamental data structures and operations on data structures in classical computer science. Among them we will cover **stacks**, **queues**, **linked lists**, and **rooted (binary) trees**. From our perspective, *stacks*, *queues*, *linked lists*, and *rooted trees* are **dynamic sets**, which are represented by simple **data structures** using **pointers**. In literature, especially in those with applications and implementations in mind, these dynamic sets are also called *abstract data types* (ADTs)

The notion of a set in computer science is the same as that in mathematics. Frequently, in mathematics, sets such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ tend to be infinite and *static*. However, there are certainly finite sets in mathematics, such as $\mathbf{Z}_5 = \{0, 1, 2, 3, 4\}$, or more advanced sets, such as *algebraic sets*, which you see in an algebraic geometry course.

Although the sets from mathematics and computer science are one and the same, the sets in computer science are manipulated by **algorithms**. By necessity, such sets tend to be finite, and perhaps more importantly, they are **dynamic**. That is, sets that are processed (or manipulated) by algorithms can grow, shrink, and generally change over time.

The change over time is the result of **operations on dynamic sets**, which can simply return information about the set without changing the set. They can equally perform an operation that will change the set from its current state to some other state.

A collection of very frequent algorithmic operations on sets can be seen below. We will consider these operations, with sometimes different names, depending on the set, as fundamental.



SEARCH(S, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Figure 1: Common operations on sets by algorithms.

Perhaps surprisingly, any of the operations listed above can be achieved in time $O(lg(n))$, where n is the size of the set S. See Chapter 13 of our book and the data structure called **red-black tree**.

1.1 Stack

The **stack** is a dynamic data structure, which is implements a **LIFO** (last in, first out) policy. The operation of "INSERT" is called "PUSH" and the operation of "DELETE" is called "POP". When implemented on a computer, the stack is implemented as an *array* or a *linked list.*

As a consequence of the "LIFO" policy, elements on the stack are handled by working with the top-most element, one element at a time **only**.

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

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

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

Figure 2: Stack operations

All three stack operations take constant time $O(1)$ to execute. Such an operation is what we termed a *primitive operation.* Note also that when the stack is **empty**, the pointer points to NIL, even though the array's indexing below starts at 1. Zero index is the default parameter of an empty stack.

As an illustration, consider the following scenario. In (a) stack S has 4 elements with the top element being 9. In (b), we see stack S after the calls PUSH(S, 17) and PUSH(S, 3). In (c) we see stack S after the call POP(S) has returned the element 3, which is the one most recently pushed element. Even though element 3 still appears in the array, it is no longer in the stack; the top is element 17. The pointer has been shifted back by one after 3 was popped.

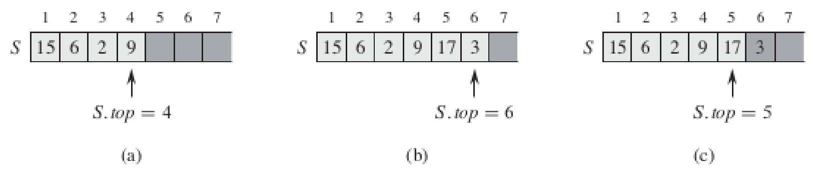


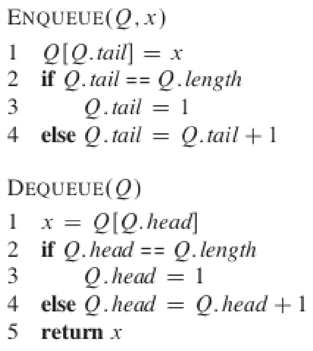
Figure 3: Stack operations

Even though these primitive operations on a stack take $O(1)$ time to execute, determining whether an element x is contained in a stack S of size n takes $O(n)$, as n elements would need to be processed in such a $SEARCH$ operation.

1.2 Queue

The **queue** is a dynamic data structure, which is implements a **FIFO** (First in, first out) policy. The operation of ”INSERT” is called ”ENQUEUE” and the operation of ”DELETE” is called ”DEQUEUE”. When implemented on a computer, the queue is implemented as an *array* or a *linked list.*

As a consequence of the ”FIFO” policy, a queue resembles a line of people, waiting to enter a store, for example. Because of this, the a non-empty queue has a **head** and **tail** element. Any time a new element joins the queue, ie. ”ENQUEUE” operation is called, it becomes the tail element. When ”DEQUEUE” operation is called, the head element only is being processed.



```
ENQUEUE($Q, x$)
$Q[Q.tail] = x$
if $Q.tail == Q.length$
    $Q.tail = 1$
else $Q.tail = Q.tail + 1$

DEQUEUE($Q$)
$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
else $Q.head = Q.head + 1$
return $x$
```

Figure 4: Queue operations

As with the stack, the operations on a queue also take $O(1)$ time. Any kind of search operation on an n element queue would take $O(n)$ time as n elements would need to be processed.

1.3 Linked List

The **linked list** data structure consists of objects which are arranged in *linear order*. While this is very similar to an *array*, where the order is typically determined by the indices, the the order in a linked list is determined by a pointer in each object.

The linked lists, of which there are several versions, provide a simple, flexible representation for dynamic sets we have been considering so far. In addition, linked lists support (though not necessarily efficiently) all the fundamental operations we defined on the very top of this lecture.

A **singly** linked list L is an object with an attribute key and one other pointer attribute, usually called *next*.

A **doubly** connected linked list L is an object with an attribute key and two other pointer attributes, usually called *prev* (for previous) and *next*.

For any element x in the list L, the x.next points to its successor in the linked list. Similarly, $x.prev$ points to the its predecessor in the list L. If $x.prev$ = NIL, the element x has no predecessor and is therefore the **first** element in L, also called the **head** of the list.

If $x.next$ = NIL, the element x has no successor and is therefore the last element in the list L, also called **tail** of the list.

An attribute (pointer) $L.head$ points to the first element in the list. A list is considered **empty** if L.head = NIL. Insertion into the linked list occurs at the location where the current $L.head$ points to.

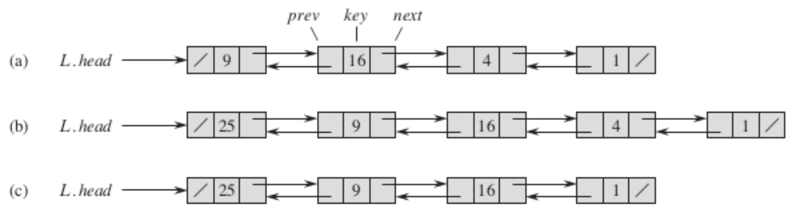


Figure 5: A doubly connected linked list

(a) A doubly linked list L representing the dynamic set of integers [9,16,4,1]. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The next attribute of the tail and the prev attribute of the head are NIL, indicated by a diagonal slash. The attribute L.head points to the head.

(b) Following the execution of *LIST-INSERT(L, x)*, where $x.key = 25$, 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.

1.3.1 Linked List Operations

Given an element x of the whose key attribute (value) has already been set, the procedure LIST-INSERT splices x onto the front of the linked list L.

```
LIST-INSERT($L, x$)
$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL
```

The procedure LIST-DELETE removes an element x from a linked list L but it must be first given a pointer to x, and it then splices x out of the list by updating pointers.

```
LIST-DELETE($L, x$)
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$
```

In order to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element x.

```
LIST-SEARCH($L, k$)
$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return $x$
```

While the operation of LIST-INSERT and LIST-DELETE take a constant time $O(1)$, the LIST-SEARCH operation takes $\Theta(n)$ time to find an element with key k in a list with n elements.

1.3.2 Circular Linked List

There is a third version of a linked list, called *circular linked list.* It is very similar to the singly/doubly linked versions. However, it differs in two ways: the head and tail are connected via a pointer and it contains a dummy object, called **sentinel**, (darkly-shaded block) which lies between the head and the tail, serving as a way to simplify the boundary condition.

While the sentinel is a proper list object, with all attributes, it carries no value associated with the key. As a result of the introduction of the sentinel, the attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$.

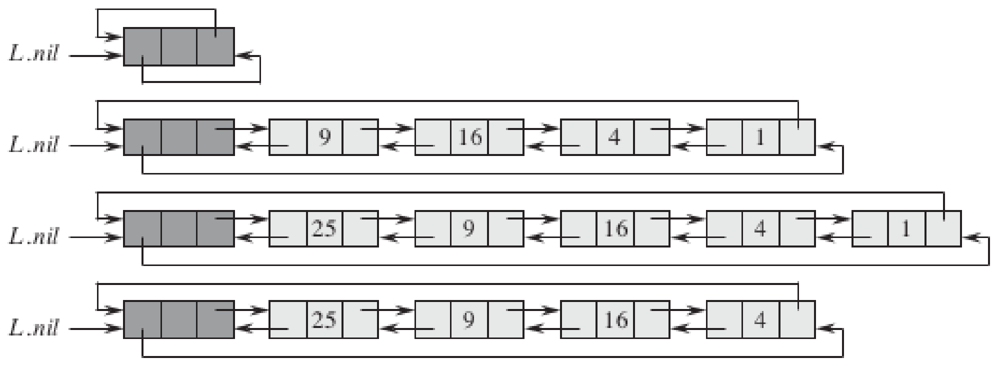


Figure 6: Circular list with the dummy sentinel shaded dark gray.

The operations on the circular list are nearly identical to those of the singly/doubly linked versions. Their time complexity is exactly the same.

2 Representing rooted trees

In this section we continue to work with the linked data structure, such as the linked list, and we will use it to represent a structure, which at face value, does not look much like a list at all. That data structure is a tree, or more specifically, a binary tree.

This approach can be extended to any tree, not just a binary tree, but for the sake of simplicity, we will focus on the binary trees as they are the most basic instance of this more general structure. In addition, binary trees, in their own right, are efficient structures, both from the time and space perspective, which are often used in the design of algorithms.

In our representation of a binary tree we will use attributes p, $left$, and $right$ to store pointers to the parent, left child, and right child for each node in a binary tree T. We will consider x to be a root if $x.p$ = NIL. If $x.left$ = NIL, then node x has no left child. Similarly, if $x.right$ = NIL, then node x has no right child.

When working with the entire tree T, we will consider the tree T to be empty if $T.root$ = NIL. If we want to point to the entire tree, we point to its root node, using $T.root$.
As an example of such a binary tree, consider the following picture

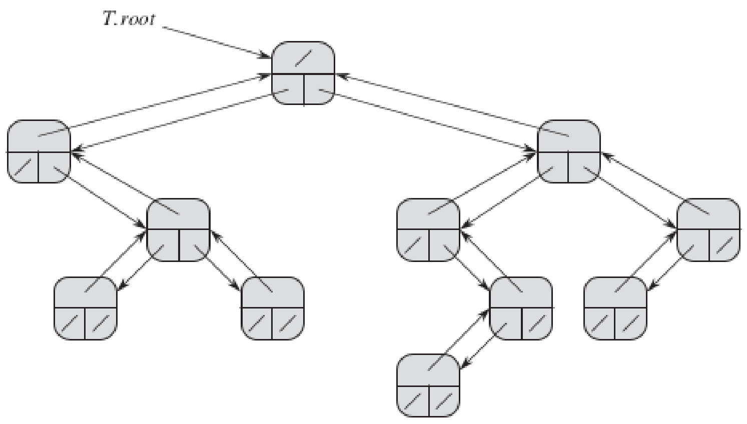


Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

As an example of a construction of such a binary tree, consider the following problem: Draw the binary tree rooted at index 6 that is represented by the following attributes

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

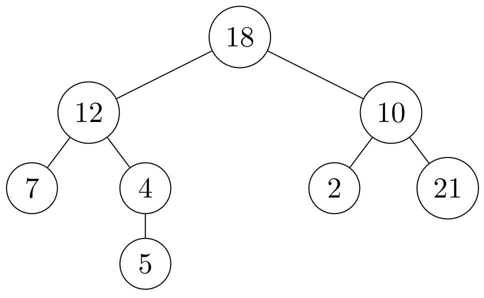


Figure 7: Resulting binary tree representation.

Arriving at the solution is straight-forward as long as we keep in mind that $left$ and $right$ refer to the index values. From key 18, the left and right children keys are located at indices 1 and 4, for example, which will result in keys 12 and 10 being added to the tree. We then continue with key 12 (index values 7 and 3) and key 10 (index values 5 and 9), and so on. Indices 2 and 8 do not appear in the resulting tree as they are not part of the tree.

Consider the following problem: Write an $O(n)$ time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

1. visit the root node (read/write)
2. traverse the left subtree in preorder (recursive call)
3. traverse the right subtree in preorder (recursive call)

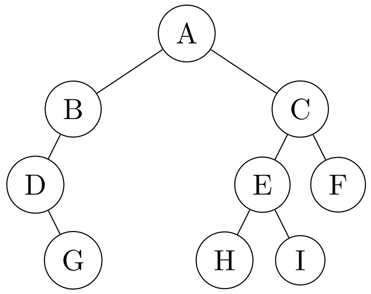


Figure 8: This kind of traversal is known as preorder traversal and results in ABDGCEHIF.

Accessing of each node (for printing) takes constant time. At each recursive call, the algorithm visits each node only once. As there are n nodes, the complexity is $O(n)$.

```
VISIT-AND-PRINT(T.root)
   if T.root == NILL then
      return
   else
      print T.root.key
      VISIT_AND_PRINT(T.root.left)
      VISIT_AND_PRINT(T.root.right)
```

Accessing of each node (for printing) takes constant time. At each recursive call, the algorithm visits each node only once. As there are n nodes, the complexity is $O(n)$.

More formally, if we label T(n) the running time of the procedure VISIT-AND-PRINT, we can derive a recurrence for the T(n). With each of the two recursive calls, we visit n/2 (left nodes) and n/2 (right nodes). This means T(n) = T(n/2)+T(n/2)+c, where c is the constant time it takes to access and print each node key. Using the Master theorem, we can show that T(n) = 2T(n/2)+c is $\Theta(n)$.