Tutorial Sheet (November 09, 2018)

ESC101 – Fundamentals of Computing

Dynamic Data Structures

Data structures are organized ways of storing data. A data structure is called *static* if its storage capacity is fixed and cannot change if demand goes up or down. A data structure is called *dynamic* if it can automatically expand and contract according to user needs.

Example 1 (Array): The array is a data structure that can store several variables of the same type (int, float, char, struct)

ADVANTAGES

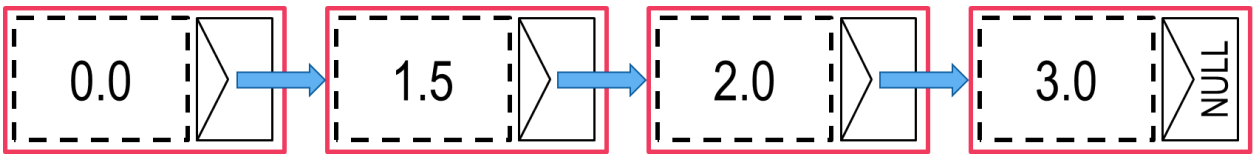
1. Extremely easy to set up an array
    a. Static array `int arr[n];`
    b. Dynamic array `int *arr = (int*)malloc(n * sizeof(int));`
2. Extremely easy to access any element. For example if trying to access i^{th} element, simply use `arr[i-1];`
3. Extremely easy to append to an array, i.e. add an element at an empty location at the end of the array. If we have `int arr[20];` but only first 12 locations are filled, then filling up the 13^{th} location with a value, say 42, us very simple. Just use `arr[12] = 42;`

DISADVANTAGES

1. Not easy, time consuming, to add elements to middle of array. If `int arr[11] = {1,2,3,40,50,60,700,800,900,1000};` then although array can store one more int, say 42, if we want to insert 42 between 40 and 50, we first need to move all elements from 50 onwards one location right. `for(int i = 4; i < 10; i++) arr[i+1] = arr[i];` Same issue with deleting an element in the middle of the array.
2. Static arrays are static data structures – cannot increase or decrease in size if demand for space goes up or down.

3. Dynamic arrays are dynamic data structures (can increase/decrease size using realloc) but the realloc process is expensive since it again involves copying elements.
4. Arrays must be stored contiguously in memory. If not enough contiguous memory available, realloc/malloc/calloc will fail even if enough memory is available but in pieces.

EXAMPLE 2 (Linked Lists): similar to arrays in that nodes of a linked list are organized one after the other. However, they are not necessarily stored contiguously in memory. For example



First node of linked list called its *head*. Last node called its *tail*.

ADVANTAGES

1. Nodes need not be stored contiguously in memory – can utilize space more effectively. Even if space is available in pieces, we will not fail to use those pieces whereas array would have failed.
2. No realloc involved – expanding or contracting a linked list does not involve copying nodes at all so very fast.
3. Inserting a node anywhere in the linked list is very easy, no need to shift around nodes. Deleting nodes from anywhere in the linked list is simple too.
4. Linked lists are dynamic data structures – can adjust to demand.

DISADVANTAGES

1. Need to write more code to set up linked list (see lecture code).
2. Not easy to access i^{th} node – need to access head i.e. the 1^{st} node, then get ask it the address of the 2^{nd} node, so on.
3. If we have address of 3^{rd} node, can go to 4^{th} node, but no way to go back to the head (1^{st} node) as no back links. Doubly linked lists (see below) solve this problem – can go back and forth.
4. More space used up in storing pointers.

APPLICATION OF LINKED LIST (STACK AND QUEUE)

Refer to the lecture code for implementation of linked list. The code implements (among other things) the following four functions

1. Node* insertAtPosition(Node *head, float x, int idx)
   Inserts a new node at position idx in the linked list (head is position 0). If head has changed, return new head else return old head.
2. Node* insertAtEnd(Node *head, float x)
   Inserts a new node at the end of the linked list. If head changes as a result, return the new head else return the old head.
3. Node* deleteAtPosition(Node *head, int idx)
   Deletes the node at position idx and returns the head
4. Node* getNodeAtPosition(Node *head, int idx)
   Returns the node at position idx (head node is position 0).

Recall from **Week 10 Mon Lab Q1**, the stack data structure which allows insertion and deletion only at the head. Recall also from **Week 10 Wed Lab Q2** that stacks can be used to evaluate mathematical expressions.

1. **Push x**: put the element x on the top of the stack
2. **Pop**: unless empty, delete element at the top of the stack
3. **Check**: unless empty, return the element at the top of the stack

```
Node* push(Node *head, float x){
    return insertAtPosition(head, x, 0);
}
Node* pop(Node *head){
    if(head == NULL) printf("Empty\n");
    return deleteAtPosition(head, 0);
}
Node* check(Node *head){
    if(head == NULL) printf("Empty\n");
    return getNodeAtPosition(head, 0);
}
```

With arrays, we needed a limit MAX on stack size. With linked lists, no such restriction. How easy is it to implement a stack using a linked list!

Recall from **Week 10 Tue Lab Q1**, the queue data structure which allows insertion only at the tail and deletion only at the head. Recall also from **Week 10 Thu Lab Q1** that queues can be used to schedule requests made to a server such as memory allocation requests.

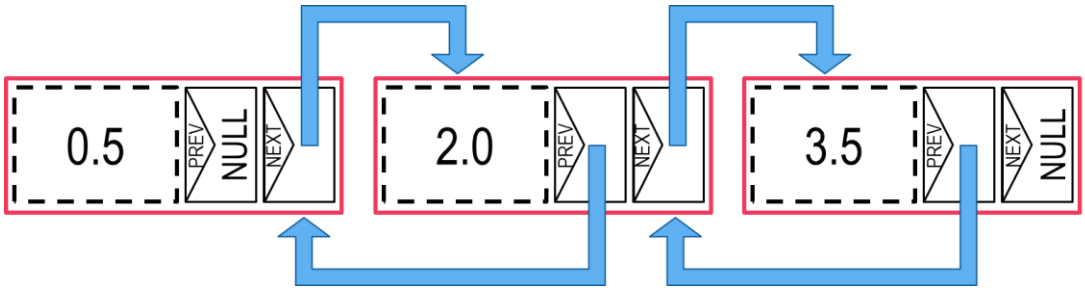
1. **Enqueue x**: insert the element x at the back of the queue
2. **Dequeue**: unless empty, delete the element at the front of the queue
3. **Check**: unless empty, return the element at the front of the queue

```
Node* enqueue(Node *head, float x){
    return insertAtEnd(head, x);
}
Node* dequeue(Node *head){
    if(head == NULL) printf("Empty\n");
    return deleteAtPosition(head, 0);
}
Node* check(Node *head){
    if(head == NULL) printf("Empty\n");
    return getNodeAtPosition(head, 0);
}
```

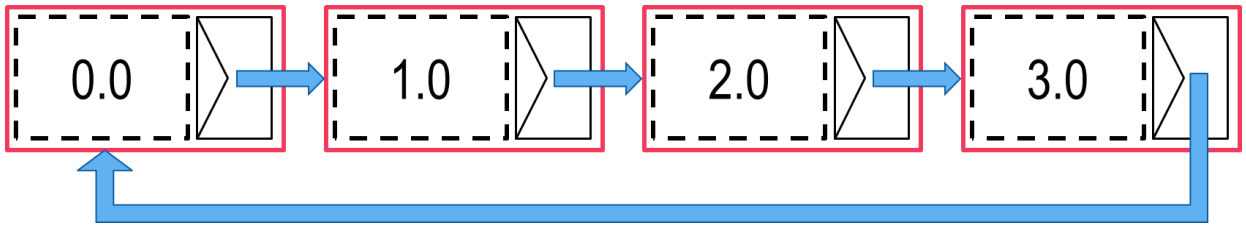
With arrays, we needed a limit MAX on queue size. With linked lists, no such restriction. Also, with arrays, needed to move things around after deleting every element, no such need here ☺.

We admit that the enqueue operation is a bit expensive here since we have to traverse the entire list from head to tail to insert at the end but even that can be sped up by having the main function maintain a special pointer which always points to the end of the linked list.

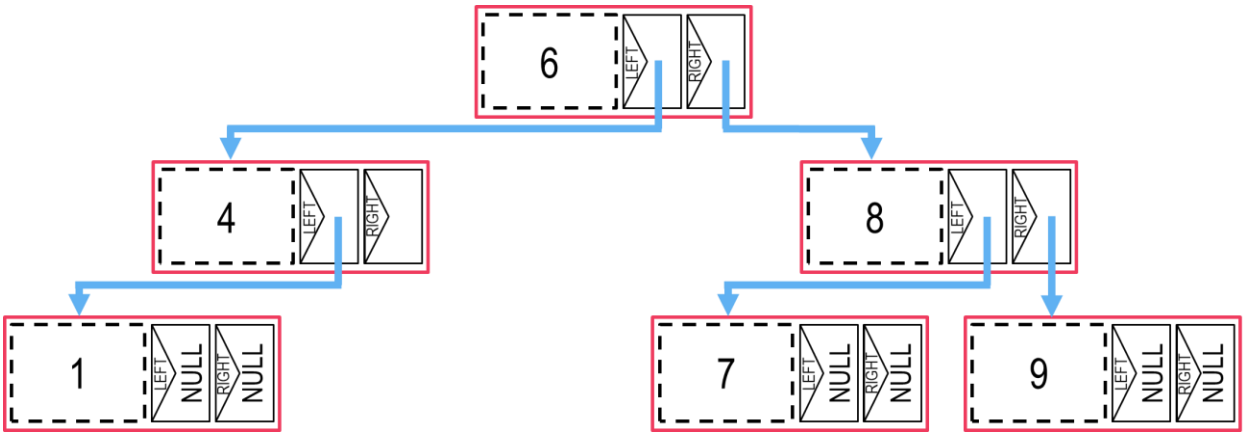
EXAMPLE 3 (Doubly Linked List): similar to linked list but all nodes point to the next node, as well as the previous node. See lecture code for implementation and usage. Can go back and forth.



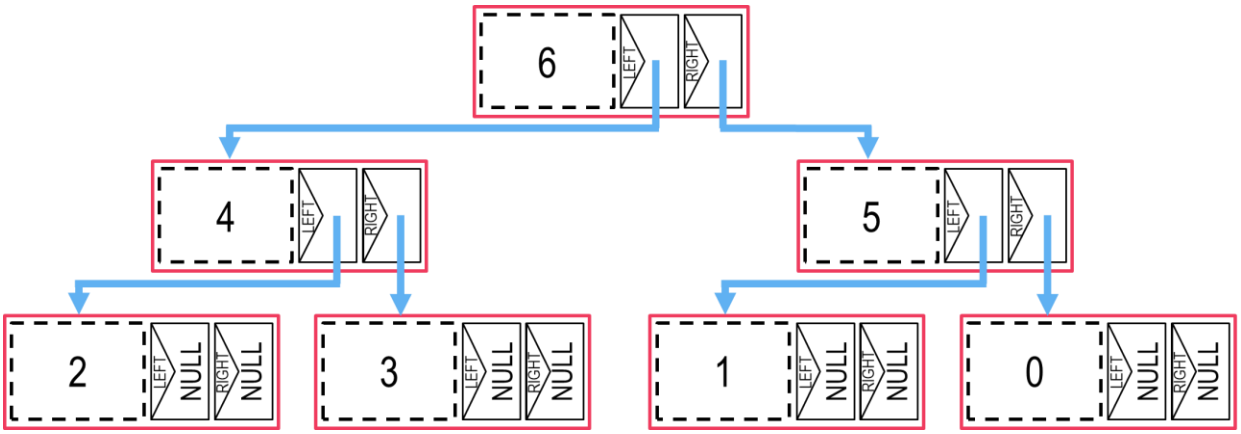
EXAMPLE 4 (Circularly Linked List): Tail node points back to head node



EXAMPLE 5 (Trees): every node of a tree has zero or more children and every node has exactly one parent except root node. Nodes with no children are called leaves. Trees where nodes can have at most two children are called *binary trees*.



The above is an example of a *binary search tree (BST)*. A BST is a binary tree if for every node, all its left descendants contain a value less than its own value and all its right descendants contain a value greater than its own value. Thus, all values in the left subtree of the root node are less than the value at the root node and all values in right subtree are greater. BSTs are named "search trees" as they help quick search.



The above is an example of a *max heap (BST)*. A max heap is a binary tree if for every node, both its children contain values less than its own value. Notice that root contains the maximum value in a max heap. In more advanced courses ESO207, CS345 etc, you will learn how to insert elements into, and delete elements from, such data structures.

ASYMPTOTIC COMPLEXITY

Suppose we have $f, g: \mathbb{R}_+ \to \mathbb{R}_+$ such that there exists a constant $c > 0$ so that for all "large" values of $x \in \mathbb{R}_+$ i.e. for all $x \geq M$ for some $M > 0$, we have $f(x) \leq c \cdot g(x)$. Then we say $f(x) = \mathcal{O}\big(g(x)\big)$ or simply $f = \mathcal{O}(g)$. Be careful that the constant c must not depend on x.

Example 1: $f(n) = 27n^2 + 4n + 6$. Then we have $f(n) = \mathcal{O}(n^2)$
Justification: For all $n \geq 4$ we have $4n \leq n^2$. For all $n \geq 3$ we have $6 \leq n^2$. Thus, for all $n \geq 4$, we have $27n^2 + 4n + 6 \leq 29n^2$. Thus, we can take $c = 29, M = 4$ to obtain $f(n) = \mathcal{O}(n^2)$.

Example 2: $g(x) = 3x^4 + 19x^3 + 165$. Then we have $g(x) = \mathcal{O}(x^4)$
Justification: For all $x \geq 19$ we have $19x^3 \leq x^4$. For all $x \geq 4$ we have $165 \leq x^4$. Thus, for all $x \geq 19$, we have $3x^4 + 19x^3 + 165 \leq 5x^4$. Thus, we can take $c = 5, M = 19$ to obtain $g(x) = \mathcal{O}(x^4)$.

Example 3: $h(y) = 2y^5 - 4y^2 + 99$. Then we have $h(y) = \mathcal{O}(y^5)$
Justification: For all $y > 0$ we have $-4y^2 \leq 0$. For all $y \geq 3$ we have $99 \leq y^5$. Thus, for all $y \geq= 3$, we have $2y^5 - 4y^2 + 99 \leq 3y^5$. Thus, we can take $c = 3, M = 3$ to obtain $h(y) = \mathcal{O}(y^5)$.

Example 4: For $f(\cdot), g(\cdot), h(\cdot)$ as defined, we have $f = \mathcal{O}(g)$
Justification: We have $g(x) \geq 3x^4$ for all $x > 0$ whereas all $x \geq 4$, we have $f(x) \leq 29x^2$. Now, for all $x \geq 4$, we have $29x^2 \leq 3x^4$. Thus, taking $c = 1, M = 4$ we get $f = \mathcal{O}(g)$.

Example 5: For $f(\cdot), g(\cdot), h(\cdot)$ as defined, we have $g = \mathcal{O}(h)$
Justification: For all $x \geq 2$, we have $4y^2 \leq y^5$ which means for all $x \geq 2$, we have $h(x) \geq y^5$. Now, for all $x \geq 19$, $g(x) \leq 5x^4 \leq x^5$. Thus, proved.

Theorem: If $f, g, h: \mathbb{R}_+ \to \mathbb{R}_+$ are three functions such that $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$, then we must have $f = \mathcal{O}(h)$ as well.
Proof: If $f = \mathcal{O}(g)$ then for some $M_1, c_1 > 0$ we must have $f(x) \leq c_1 \cdot g(x)$ for all $x \geq M_1$. Similarly, if $g = \mathcal{O}(h)$ then for some $M_2, c_2 > 0$ we must have had $g(x) \leq c_2 \cdot h(x)$ for all $x \geq M_2$. Taking $M = \max\{M_1, M_2\}$ and $c = c_1 \cdot c_2$ gives us $f(x) \leq c_1 \cdot g(x) \leq c_1 \cdot c_2 \cdot h(x) = c \cdot h(x)$ for all $x \geq M$. Thus proved.