# Data Structures

Seonghyun Park

dd November, 2018

# Outline

# Today's Goal

- Understand how to create dynamically allocated data structures e.g. *linked list* and *binary tree*.
- Practice allocating and deallocating memory in the data structures.

# Data Structures

- Which data type would you use if you have to store information of students who registered to a course?
  - Array? You may need more...
- What kind of things you need to think about when choosing a data structure?
  - Performance? Memory usage?

# Linked list

- Linked list is a linear data structure, where each node contains the address of (i.e. pointer to) the next node.
- Many variations exist, including doubly linked list, or circular list.

```
struct Node {
  int val;
  Node* next;
}
```

# Linked List Find

- Find node $N$ with value $v$ in the linked list $\mathcal{L}$.
  - Let *iter* be the address of $\mathcal{L}$.
  - Advance *iter* (i.e. `iter = iter->next`) until a node with value $v$ is found.

# Linked List Insertion

- Insert a node next to a given node *N*.
    1. Store address of the next node of *N* into *Old*.
    2. Create a new object *New* and its next node becomes *Old*.
    3. Substitute *New* for *Old* as the next node of *N*.

```
void insertNext(Node* node, int val)
{
  auto oldNext = node->next;
  auto newNext = new Node{val, oldNext};
  node->next = newNext;
}
```

# Linked List Deletion

- Delete a given node $N$ from the linked list $\mathcal{L}$.
  1. Find $P$ in the $\mathcal{L}$, whose next node is $N$.
  2. Set the next node of $P$ to be the next node of $N$.
  3. Deallocate memory pointed by $N$.

```
bool deleteNode(Node* head, Node* node)
{
  auto p = prevNode(head, node);
  if (!p)
    return false;
  p->next = node->next;
  delete node;
  return true;
}
```

- We may do better if we keep the pointer to the previous node!

# Stack

- *Stack* has limited interfaces `push(stack, value)` and `pop(stack)`.
- `push()` stores `value` to the stack.
- `pop()` removes a value from the stack.
- Latest value added to the stack will be popped first. (LIFO)
- The underlying implementation may use either an array or a linked list.

```c
struct Stack { ... };
void push(Stack* stack, int val);
int pop(Stack* stack);
```

# Queue

- *Queue* has limited interfaces `enqueue(q, v)` and `dequeue(q)`
- `enqueue()` stores value to the queue.
- `dequeue()` removes a value from the queue.
- Oldest value added to the queue will be dequeued first. (FIFO)
- The underlying implementation may use either an array, a linked list or two stacks. (How?)

```
struct Queue { ... };
void euqueue(Queue* queue, int val);
int dequeue(Queue* queue);
```

# (Optional) Binary Tree

- Linked list has a linear structure. In many cases we should perform linear search.
- Linear search is not bad. Can we do better? Use *Binary Trees*!
- Each node in the binary tree has two children nodes.

```
struct Node {
  int val;
  Node *left, *right;
};
```

- A binary called a *Binary Search Tree*, if it satisfies this property:
  - $\forall N \in \mathcal{T}$, $N \to \mathit{left} \to \mathit{value} \leq N \to \mathit{value}$ and $N \to \mathit{value} \leq N \to \mathit{right} \to \mathit{value}$

# (Optional) Binary Search Tree Insertion

```
void insertLeaf(Node* root, int val)
{
  auto iter = root;
  auto leaf = new Node{val, nullptr, nullptr};
  while(true) {
    if (val < iter->val) {
      if (!iter->left)
        iter->left = leaf; break;
      iter = iter->left;
    } else {
      if (!iter->right)
        iter->right = leaf; break;
      iter = iter->right;
    }
  }
}
```