(01) Lists - array

Let's think about an array, and what kind of operations we might want to perform in the real world.

Say, we have a sorted array.

10	12	16	17	18	21	22	

n = current number of elements in the array

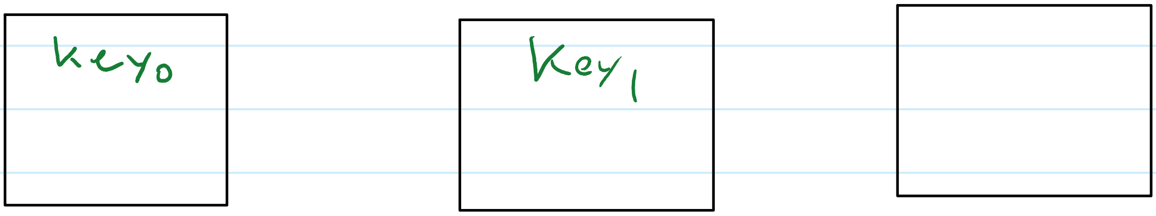
Insert new element: e.g.

insertElement(5)

- worst case scenario peformance wise?

(02) The Linked List

- Alternative method to the array for storing a list of data in the memory.
- Each element occupies its own "node"
- Unlike the elements in an array, the nodes are not sequential in the memory space
- We can add a new node anywhere in the list with no need to move other nodes.
- Traversing a LL is done by "hopping" around in the memory space by following pointers.
- Besides the pointer that is a apart of each node, we have a separate pointer variable that is typically named "head"



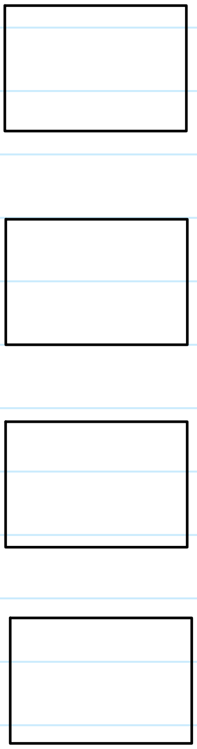
(03) The Linked List example

Example: dynyamically create a linked list to store elements of an N=4 length array. The head node should contain the 0-index element of array, the next node should contain the 1-index element of array, and so on.

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

int main(){
   int arr[] = {5,12,14,17};

   return 0;
}
```



(04) Abstract Data Type

For each data staructure, we will usally first define the ADT

- A collection of member data and the allowed operations on that data.
- Abstract, because the user (i.e. the programmer using the the class), only has info about the inputs, the outputs, and the explanation of the actions.
- Can think of it as a pseudo-code class definition.
- Not language specific

Singly Linked List Generic ADT:

- private:
  - head - ptr to first element in list. nullptr means the list is empty
- public:
  - initialize() - set header to nullptr
  - nodePtr = search(value) - find a value and return ptr to its node
  - insertNode(previousPtr *(could be string type)*, newKey(s))
    - ▪ given the *previous* node, create a new node and insert it after *previous*
    - ▪ if user desires to enter new node at the head of the LL, they should call function w/ specific value (e.g. nullptr or "")
  - displayList() - traverse the LL (starting at head) and print contents
  - deleteNode() - remove node from list and re-link list
  - deleteList() - clear the entire list

General rule to keep in mind:

Every method in the ADT should be designed such that once it performs its task, the integrity of the data structure is preserved.

(05) The Header File

```
// SLL.hpp - interface file (header file)
#ifndef SLL_H
#define SLL_H
struct Node{
    string key;
    Node *next;
};
class SLL{
private:
    Node* head;
public:
    SLL(); // constructor declaration
    ~SLL(); // destructor declaration
    Node* search(string sKey);
    // Precodndition: sKey parameter is a string type
    // Postcondition:  if found, returns a pointer to the node containing sKey value.
    // If not found, returns a null pointer.
    void displayList();
    // Precondition: the head node is defined.
    // Post condition: display the key values of the entire list, starting with
    // first node and ending with last node.
    void insert(string afterMe, string newValue);
    // Precondition: afterMe is a valid pointer to a node in the linked list.
    // newValue is a valid string.
    // Postcondition: a new node is created and newValue is stored as its key.
    // The new node is added after node containing afterMe.
    void deleteNode(Node* deleteNode);
    // Precondition: head and tail pointers are set.
    // Post condition: node where with a matching key value is deleted.
};




#endif
```

(06) Implementations in C++

Your workflow when implementing a class should follow some logic that will allow for intuitive checkpoints along the way.

Where to start?

Order of implementations:

1. Constructor
2. Search
3. Insert
4. Display
5. delete

```
SLL::SLL(){ // constructor definition
    head = nullptr;
}
```

(07) Implementations in C++ (search)

```
Node* search(string sKey);
// Precodndition: sKey parameter is a string type
// Postcondition:  if found, returns a pointer to the node
// containing sKey value.
// If not found, returns a null pointer.
```

Approach:
Starting at the head, use a local pointer to *crawl* down the list until a node with matching key is found.

```
Node* SLL::search(string sKey){
    Node* crawler = head;
    while( crawler != nullptr && crawler->key != sKey){
        crawler = crawler->next;
    }
    return crawler;
}
```

(08) Implementations in C++ (insert)

Must account for different scenarios:

1) Empty list
2) List is not empty. User wants to insert node at the beginning of list (thus making it the new **head.)**
3) List is not empty. User wants to insert a node specifying the preceding node.

```
void SLL::insert(string afterMe, string newValue){

}
```

Example:
afterMe = "hello"
newValue = "there"

(09) The Destructor

Just like there is a constructor that gets called automatically when an object is instantiated, the destructor gets called when its function pops off the stack.

No need to define destructors when not working with dynamic memory. (Get defined by default.)

A data structure class that uses explicit dynamic memory allocations (**`new`** keyword), a destructor should be defined.

The syntax in definition uses the ~<class name>. E.g.:

```
class SLL{
private:
       ...
public:
   SLL();    // constructor
   ~SLL(); // destructor

}
```

We can also call the destructor manually, but this is NOT typically done. E.G.:

```
int main(){
   SLL S0;
   S0.~SLL(); // Just FYI

}
```

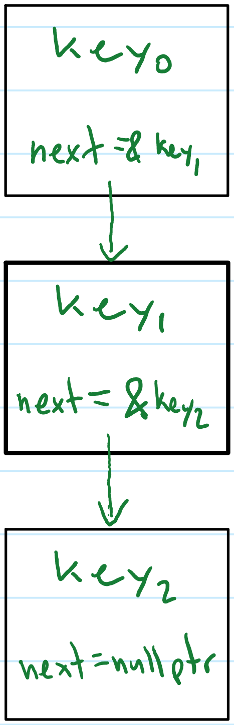
destructorDemo

https://comicvine.gamespot.com/destructor/4005-32088/

(10) Implementations in C++ (destructor)

Approach:
Set the head to head->next, then delete old head. Keep going until end of list.

```
SLL::~SLL(){
    Node* crawler;
}
```



(11) Implementations in C++ (delete)

Remove node from LL pointed to by ptr.

Function should never be called with null ptr.

2 cases to consider:

1) Node to be deleted is head
    a. establish new head (head->next)
    b. deallocate old head

2) Node to be deleted is somewhere else in the list.
    a. traverse list, stop at node previous to the one to be deleted
    b. reconnect the previous node to the next node
    c. deallocate the node to be deleted

(12) Implementations in C++ (delete)

Remove node from LL pointed to by ptr.

Function should never be called with null ptr.

```
void SLL::deleteNode(string deleteKey){
}
```