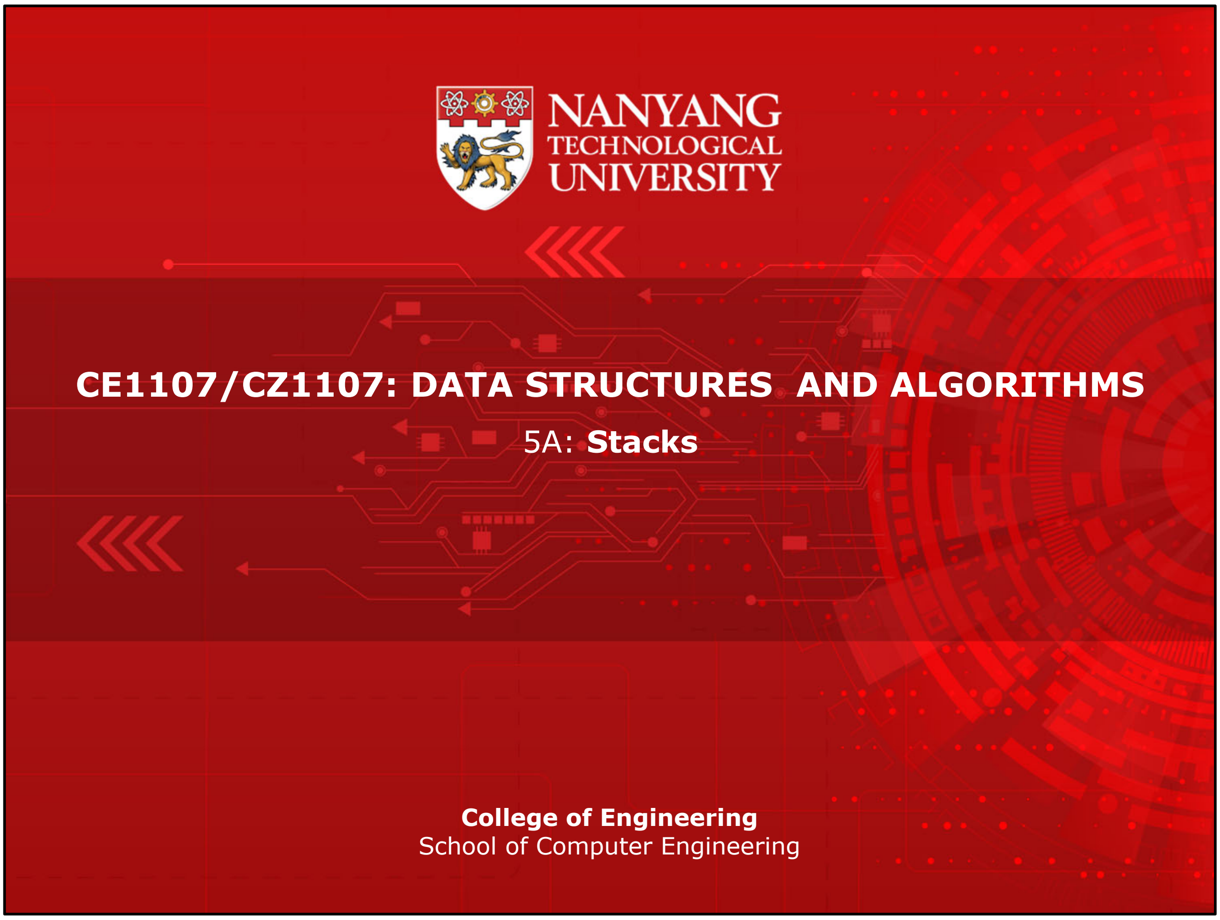


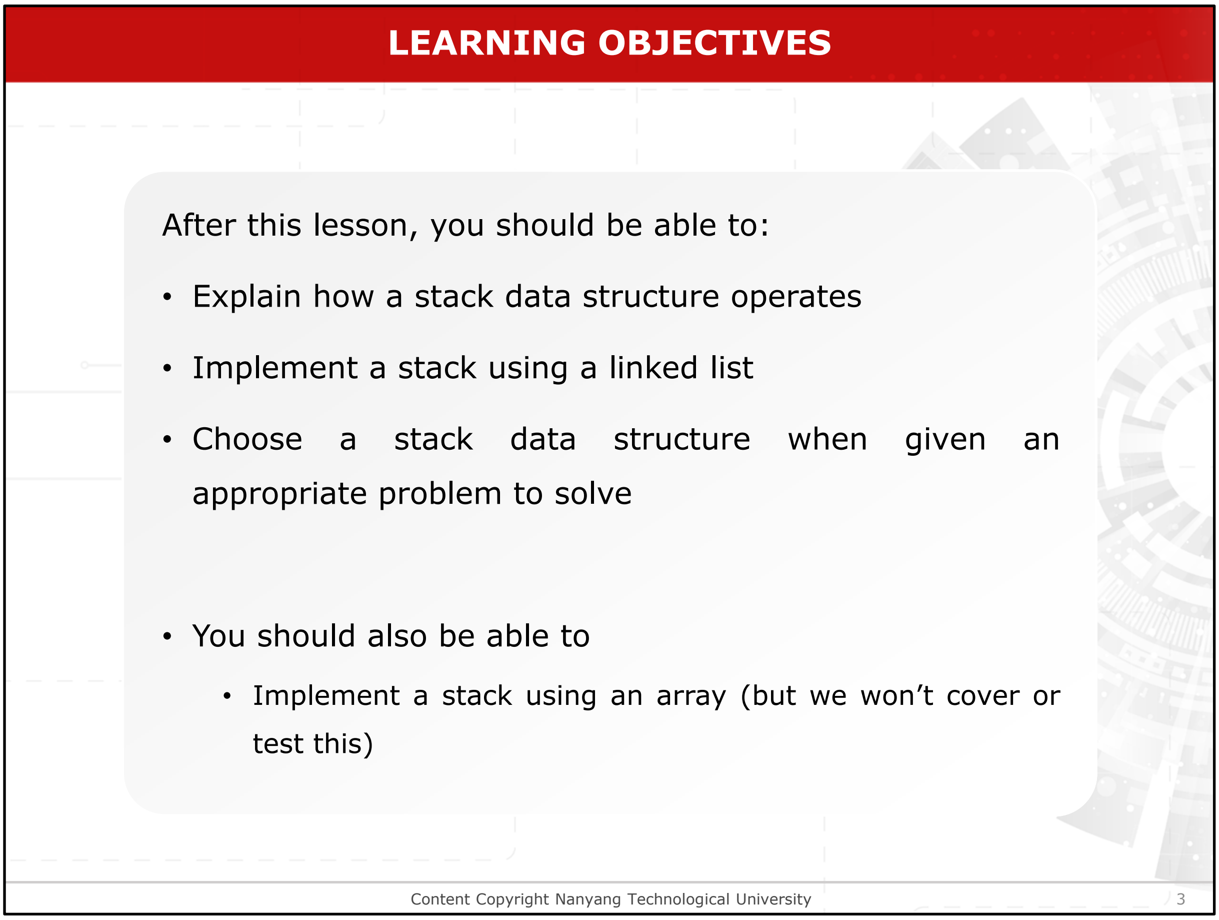
Today, we will cover two additional linear data structures that are built on top of the foundation of linked list. I have separated the materials as stacks and queues but you will notice that most of the slides are repeated or look very similar.

TODAY

- Motivating application
- Stack data structure
- Stack implementation using linked lists
- Stack functions
    - push()
    - pop()
    - peek()
    - isEmptyStack()
- Working examples: Applications

 2

We will run through a simple motivating application which will help us to learn additional data structures. We will learn the concept and the implementation of stack data structures. The basic functions of stack will be discussed first and I will explain how we use stacks and why we might need to use stack in the first place by using a sample application.



At the end of this lecture, you should be able to:

- Explain how a stack data structure operates
- Implement a stack using a linked list
- Choose a stack data structure when given an appropriate problem to solve

You also should be able to implement stack using an array but we will not cover or test this.

MOTIVATING APPLICATION

- Scenario:
  - Container yard with 10x10 grid layout, and each square has a stack of containers
  - Overhead crane can move over any stack
  - Crane can hold/inspect one container at a time from top of stack and move it to any other stack
  - Grid square at (0,0) is empty
- Task: Simulate the operation of this container yard

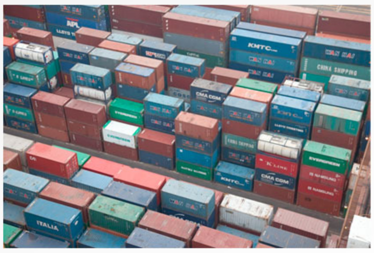


Image credits: http://kaidashton.blogspot.sg/2012/03/hong-kong-container-yards.html, http://es.made-in-china.com/co_goldhorse/product_RTG-Rubber-Tyre-Gantry-Crane-of-Port-for-Container-40FT-or-20FT-Payload-40-Ton_hhruniseg.html

This example was taken from the 2002 exam paper. Here, we have a container yard with ten by ten grid layout.

If you look down from above, it is ten by ten and each of the grid squares represent a stack of containers. Therefore, the yard has ten rows, ten columns and each stack can be up to ten containers.

The overhead crane can pick one container at a time, inspect the ID on the container and move to anywhere else. When we need to shuffle things around, we might need a temporary working space. For an example, one grid square at (0, 0) left empty for you in case you need to do some temporary operations such as shuffling containers.

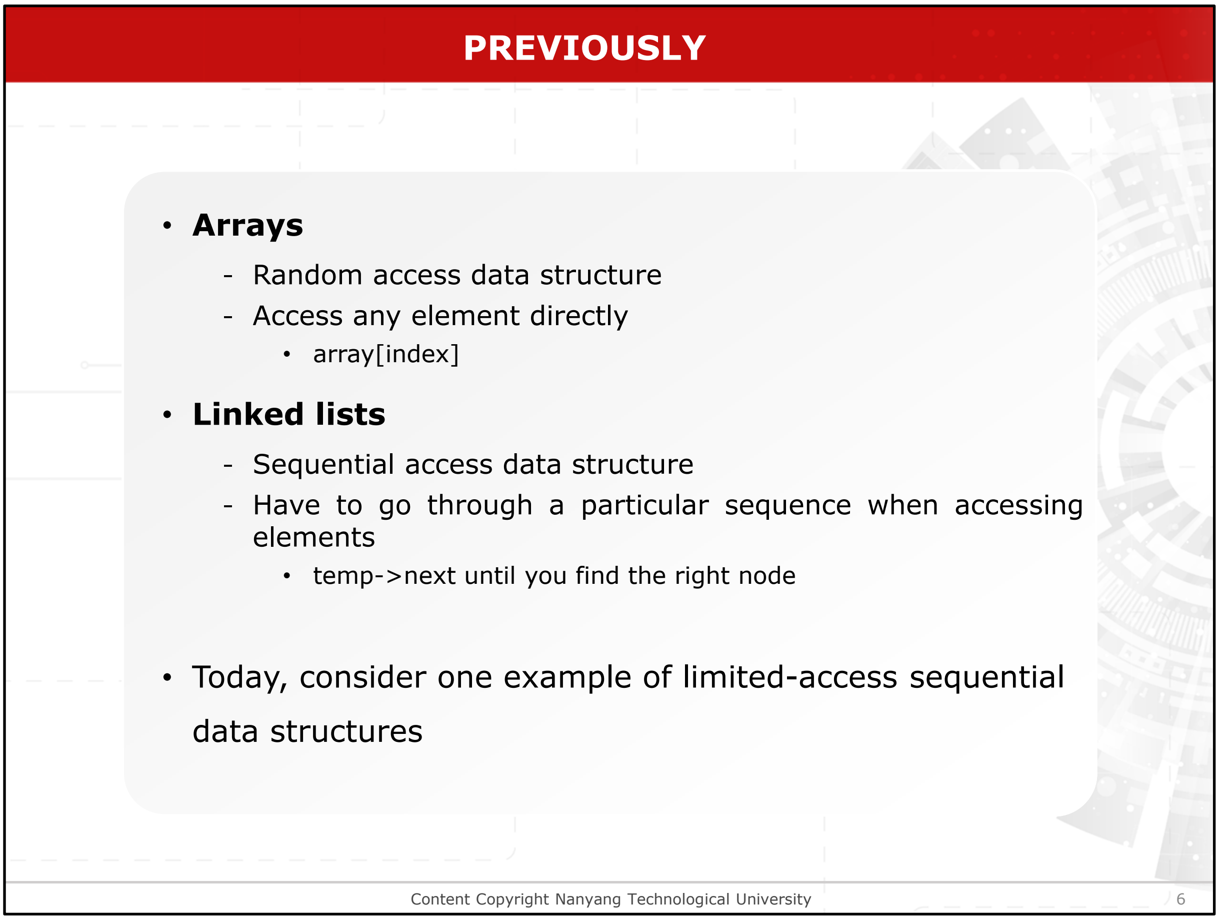
So, let’s figure out how to simulate these kind of operations. How to write a program that allows us to store the containers and stack of containers inside a memory in a related structure and call a certain function that can move the crane over, search a container, move a container to another stack, etc.

TODAY

- Motivating application
- **Stack data structure**
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

 5

Before we move to the implementation, we will learn what stack data structure is because it is quite intuitive.

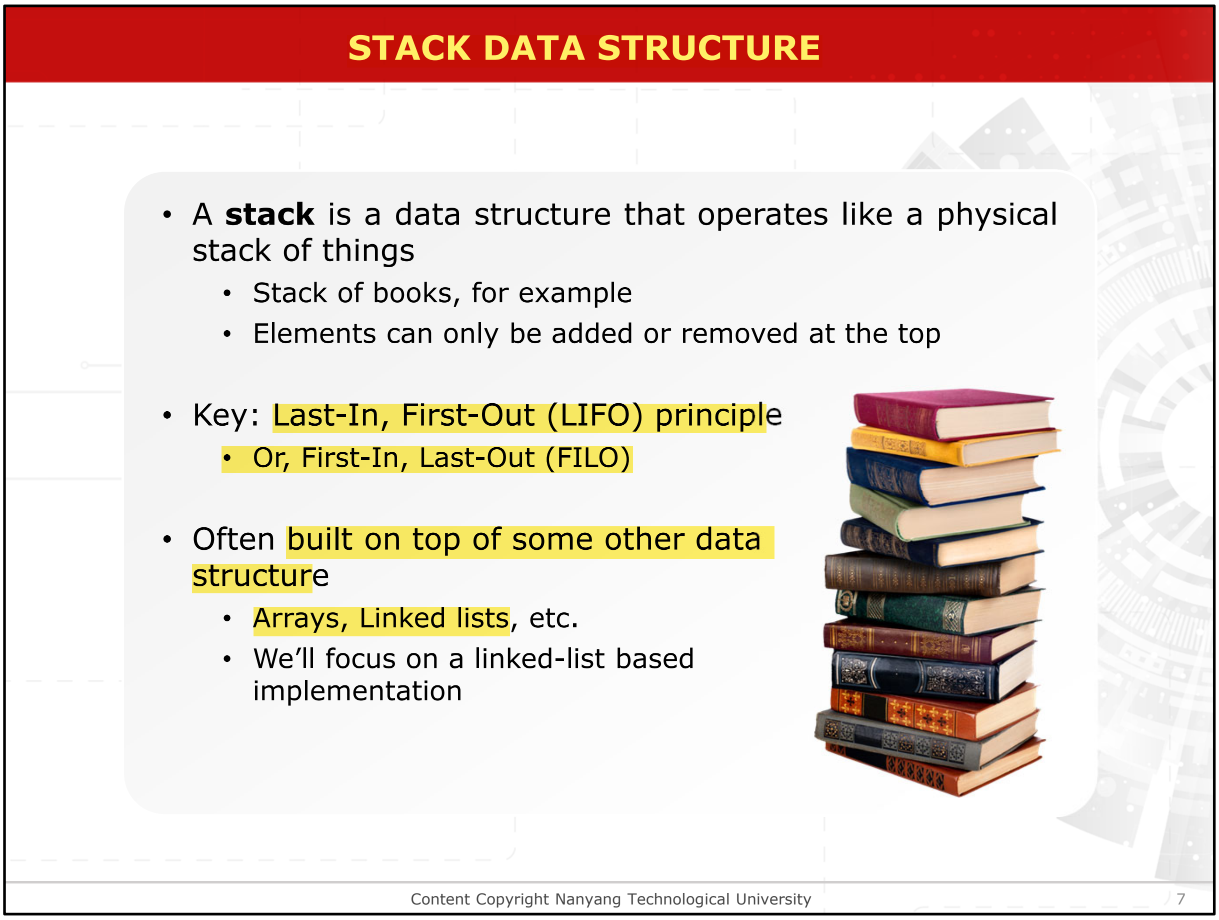


In previous lectures, we talked about linked list structures and compared linked lists with arrays. An array is a sequence of contiguous elements that are squished together. It is a random access data structure. Therefore, once you access the array and when you know what exactly you want, you can straightaway jump into that element. But linked list does not have random access property. Because of the way linked list is created, I cannot straightaway jump to the desired node.

In linked lists, we have to use sequential access method to get to the desired node. That is why we discussed about inefficient operations because the longer the linked list gets, the worse the efficiency is.

In the previous lecture, we have discussed how the difference between sequential access and random access leads to the strengths and weaknesses of arrays and linked lists, and at what situations you should choose them. Today, we will discuss about sequential access data structures and how we can constrain it further.

In previous lectures, we learned about insert node and remove node functions. But, instead of removing a node from anywhere, sometimes you may want to constrain it in such a way that we insert or remove only in certain positions. We call this limited access linear sequential data structure.

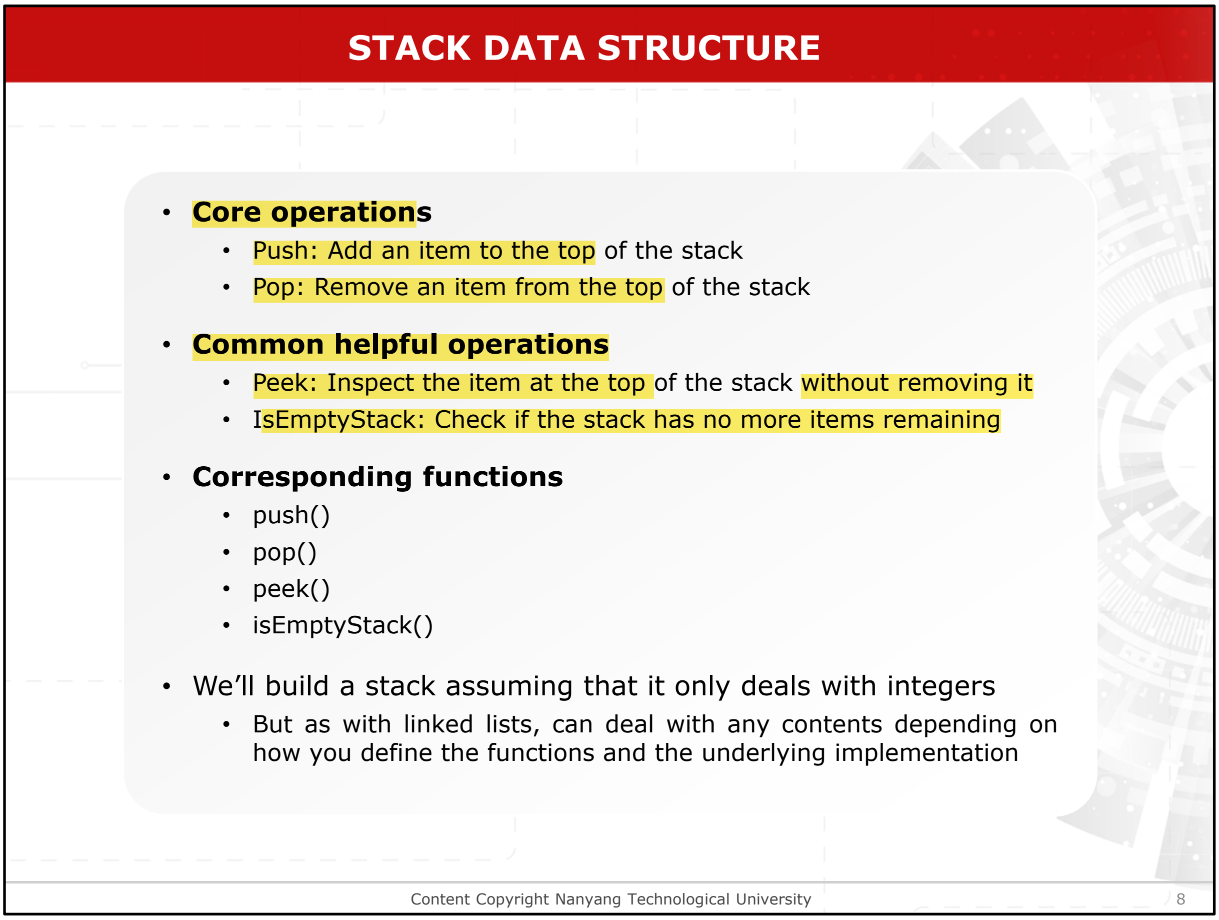


A stack is a data structure which operates like a physical stack of things. For an example, if you have a stack of books, there are only certain places to where you can add a new item. You can only add a book to the top or remove from the top of the stack. Though there are multiple ways to add or remove a book from anywhere of a book stack, if I want to easily remove or insert one item, it should be from the top of the stack.

The key thing of this constrain is that we can be ended up with last in first out principle sometimes. This is because the first item you put in to the stack is at the bottom of the stack. Therefore, to remove items from the stack I need to start with the top most item.

We got the first in last out or last in first out because we constrain the position where we can access in this stack. Then we get to the implementation. Usually, it builds on top of other data structures. Therefore, instead of worrying about things like maintaining the order of the items, or special cases such as when the stack is empty and we need to insert an item.

You have already learned insert node and remove node functions that covered all these special cases. Therefore, we are going to use these functions to implement a stack and we will use all linked list functions to do things like, add items, remove items, find the number of items, etc.



Same as in linked lists, we will insert and remove items from a stack but since we are restricting the position where the insert and remove items are placed, we will use a special name for it.

In stacks, for these operations we call push and pop. In push, we add an item to the top of the stack. And in pop, we remove an item from the top. These are the only two operations that allow you to modify data in a stack.

The peek function allows you to look at the top most item in the stack without actually removing it. If you think of a book stack, you may recall that you can look at the entire stack of books because by looking at the spines you can see the titles.

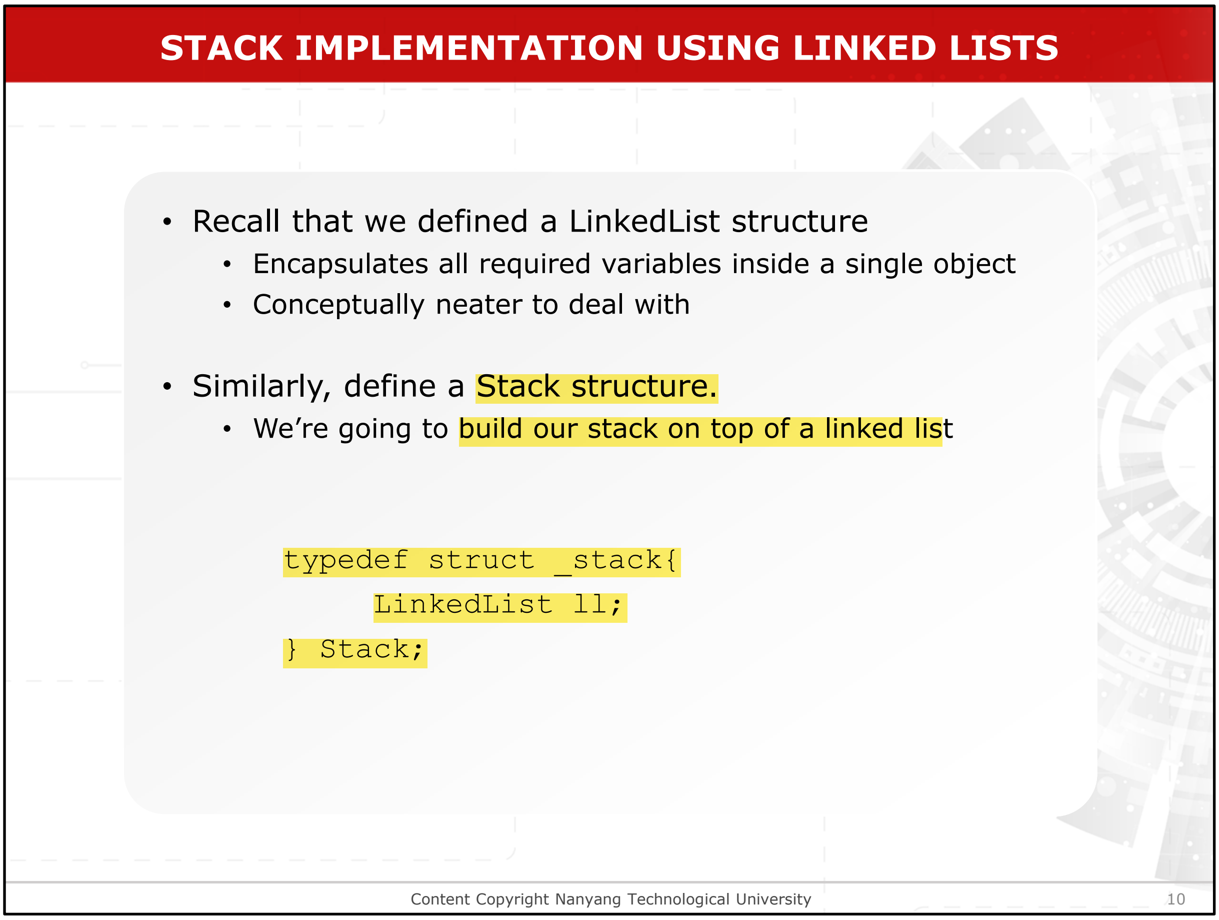
But since this is a computer program, you can only see the value by accessing the variable. You will see why peek function is helpful in future.

Empty stack is also a simple function which checks whether the stack is empty. It is helpful for special cases where we do not care how many items are left inside a stack, but care if it is empty or not.

TODAY

- Motivating application
- Stack data structure
- **Stack implementation using linked lists**
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

In this part we will look at the actual implementation of stacks. We will use linked list struct to implement the stacks.

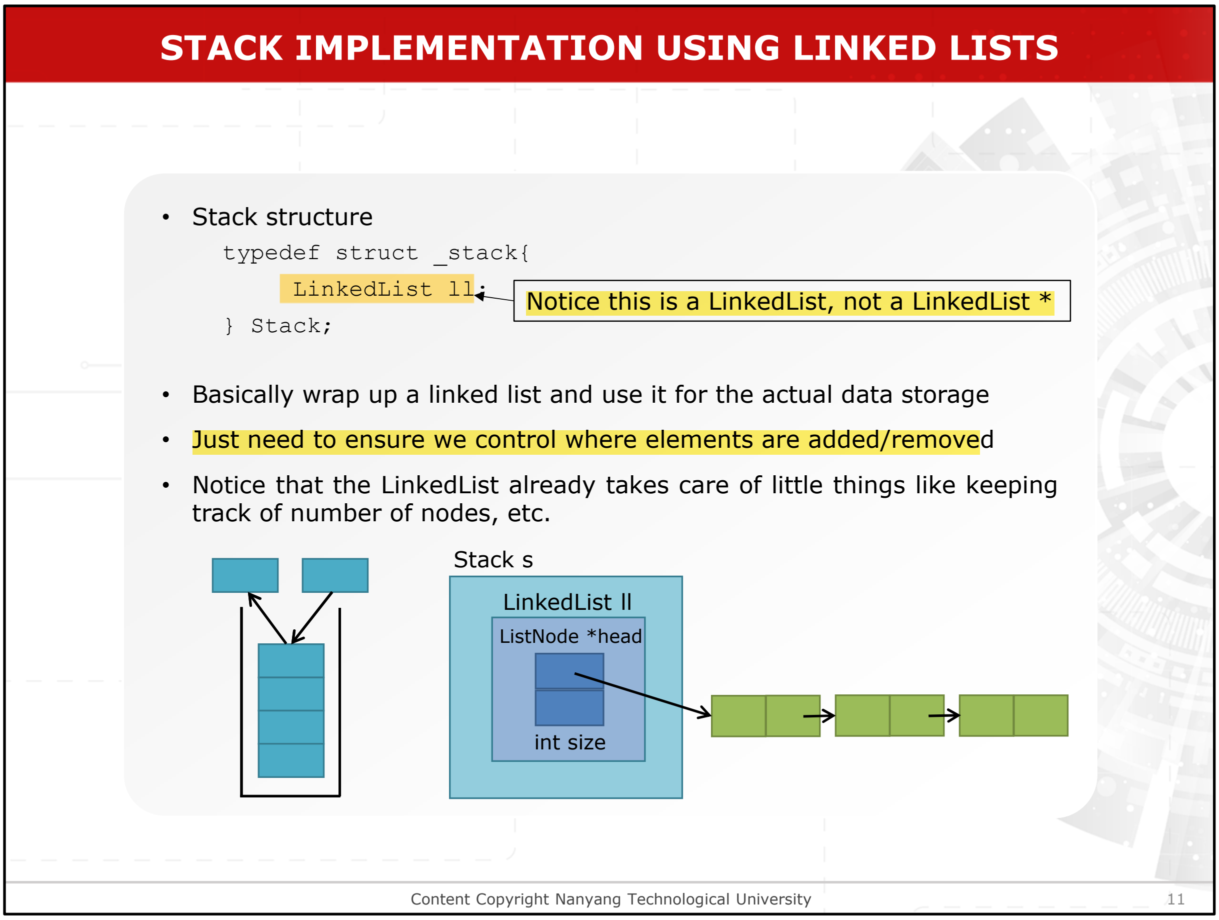


If you remember how painful it was to understand the pass in pointer to the head pointer concept, you may recall that we learned why it is better to go with linked list structure that encapsulates all the parts of the structure.

Earlier, we used linked list structure to wrap up all the linked list items including the head pointer and size integer variable. Now, we use the linked list structure and warp it again inside a stack data structure.

We are going to take a copy of the linked list struct and throw it into a stack struct. This is a very simple struct. Inside the stack struct it has an actual linked list struct. This is not a pointer to a linked list.

This is useful, because we have all the functions and variables we need inside this linked list struct. Therefore, we do not need to rewrite the codes.



Now the previous diagram has changed. We have an outer box for stack that covers the linked list struct. We are trying to build the concept that is visualized in the left diagram.

A stack has the values that are come in and go out only at the top position of the stack. But if we look at the memory layout, at the end we will get a stack structure. Inside the stack struct we have the linked list struct and inside the linked list struct we have two items as the head pointer and the size variable.

This is the simplest form of linked list we can have which does not include doubly linked nodes, or tail pointers. We have the nodes outside the struct and connected to the struct via the head pointer. The nodes are defined as external blocks of memory. Therefore, now we have a stack struct which wraps up the linked list structure which warps up a head pointer and a size variable.

In this way we do not need to worry about rewriting the code. For an example, if we want to add an item to the stack we have to call the insert node function in the linked list.

TODAY

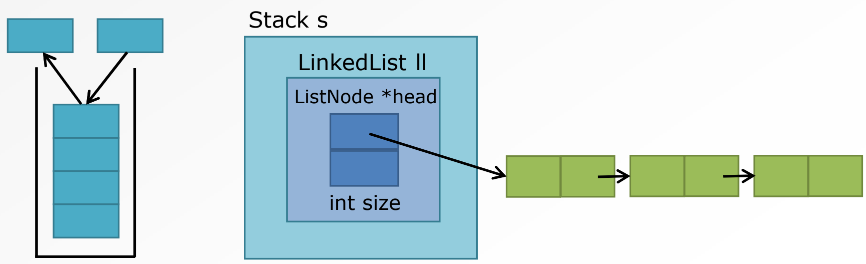
- Motivating application
- Stack data structure
- Stack implementation using linked lists
- **Stack functions**
    - push()
    - pop()
    - peek()
    - isEmptyStack()
- Working examples: Applications

 12

Stack functions.

STACK FUNCTIONS: push()

- Push() function is the only way to add an element to the stack data structure
- Only allowed to push() onto the top of the stack
- Question:
  - Using a linked list as the underlying data storage, does the first linked list node represent the top of the bottom of the stack?



 13

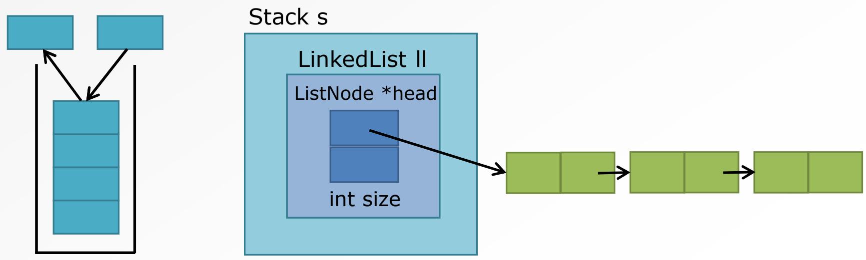
We have two core functions in stacks as push and pop. The push function is the only way you can use to add an item to the stack.

The first thing you need to figure out is how your linked list is arranged. Does the first linked list node correspond to the top position or the bottom position of the stack? This is important because the linked list has certain operations that are efficient and certain operations that are inefficient when the size of the linked list increases.

Therefore, we need to choose a mapping where adding an item to the stack is as efficient as possible.

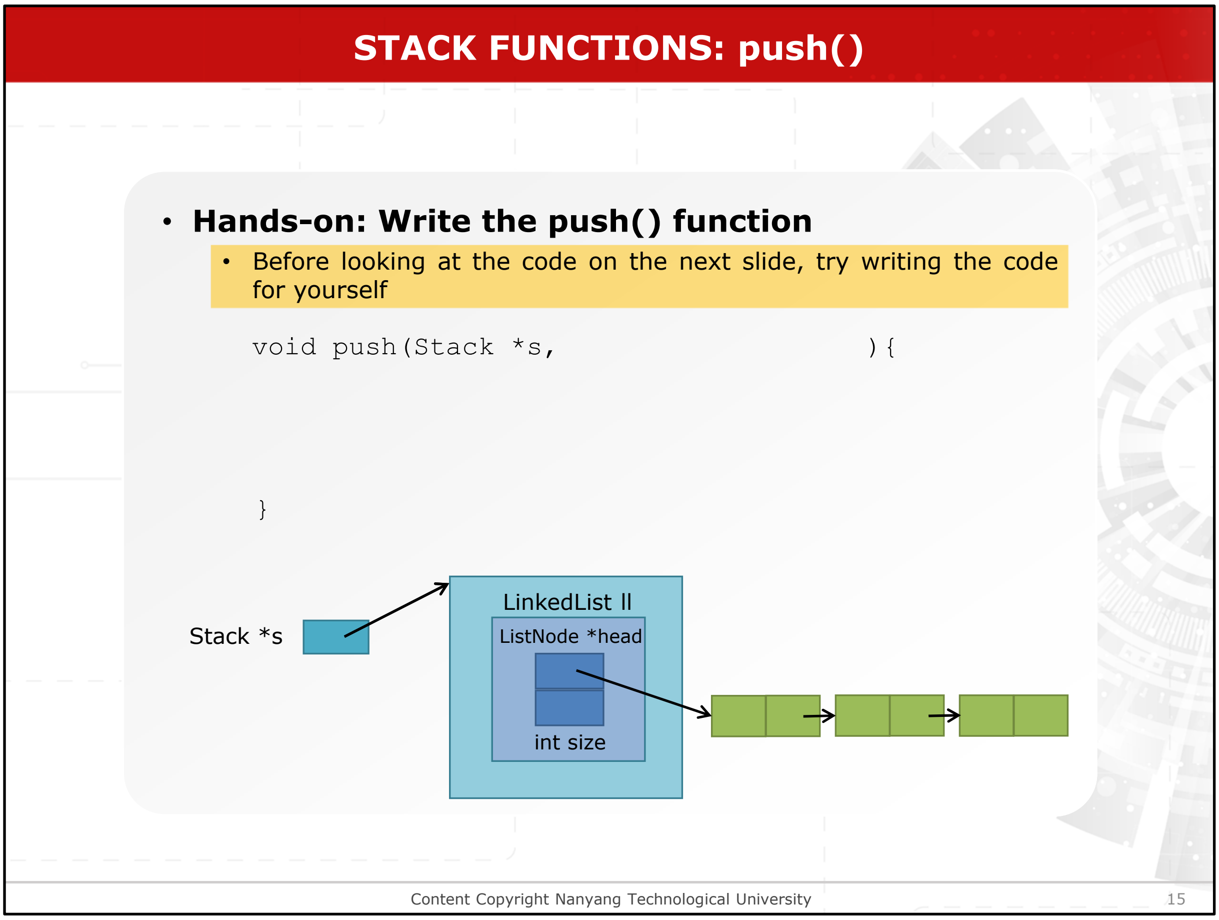
STACK FUNCTIONS: push()

- **Hands-on: Write the push() function**
  - Define the function prototype
  - Implement the function
- Answer is a few slides down, so don't look yet
- Requirements
  - Make use of the LinkedList functions we've already defined
  - Insert a new <u>integer</u> (what data type for the "item"?)
  - Insert at the <u>top</u> only (what index position?)



 14

Here, I want to first try to write the push function by yourself. You are going to use the functions that you have learned so far. I have already given you a partial function prototype.



Try to avoid looking at the next slide because that contains the answer. Try to figure out what parameters you need and how build the code so that it can add a new item to the stack.

If you are wondering what is stack *S, remember we are passing the stack structure by reference. Which means we have a pointer so that the actual stack can be accessed from inside of the push function.

Now, the first question is, what corresponds with the top of the stack? Is it the first node or the last node of the linked list? We need to avoid traversing the linked list all the way to the end. Therefore, it is obvious that the first node corresponds with the top of the stack. Every time we access the stack, we access the top most position of the stack. Therefore, now all we need to do is get the head pointer which takes us to the first node every single time.

If you chose the opposite direction where the last node corresponding with the top of the stack, every time we access the top of the stack, we need to jump to the very end of the linked list, which is not efficient.

STACK FUNCTIONS: push()

- First linked list node corresponds to the top of the stack
- Last linked list node corresponds to the bottom of the stack
- Pushing a new node onto the stack → adding a new node to the front of the linked list

```
void push(Stack *s, int item){
    insertNode(&(s->ll), 0, item);
}
```

  - Notice that this is a very efficient operation
- We can also add the new nodes to the end of the linked list
  - Need to use a tail pointer to make the operation efficient

 16

In the insert node function, we needed the index position to add the value to any position of the linked list. But here, we do not need to worry about the index position because we are not allowed to add or remove from any place other than the top of the stack. So, we actually have to care about which stack we are operating on and what is the value we are trying to add.

What we realize is that we are trying to actually add a new item, a new node to the linked list. This linked list is supposed to store all of the items on the stack. The first node of the linked list corresponds with the top of the stack. Therefore, when we want to add an item to the top of the stack, we are essentially inserting a new node at the front position of the linked list which is at index 0. We also do not need to worry about malloc or adjusting head pointer since insert node function takes care of it.

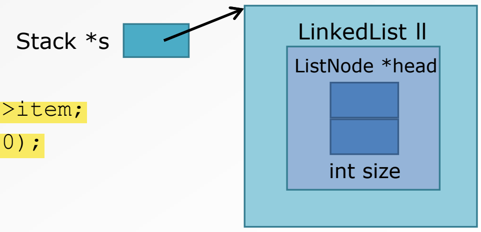
Since we are trying to access the linked list, we need to pass the linked list into the insert node function. Therefore, by using s->ll, the stack s gets hold of the linked list. Now, to pass in the address of the linked list to get to the linked list struct, we use the ampersand around the entire block of s->ll. Then, we pass the struct by reference into the insert node function.

The rest of the code is self-explanatory. If we want to flip the order, we can make use of the tail pointer. We will get back to it later.

STACK FUNCTIONS: pop()

- Popping a value off the top of the stack is a two-step process
  - Get the value of the node at the front of the linked list
  - Removing that node from the linked list

```
int pop(Stack *s){
    int item;
    item = ((s->ll).head->item;
    removeNode(&(s->ll), 0);
    return item;
}
```



- Need a temporary int variable to hold the stored value because I can’t get it after I remove the top node

Pop function is a bit more complicated. Here, we use the remove node function. The only difference from the push function is that we need to return a value.

We actually need the removing value to be stored unless somebody else can figure out a way to simultaneously remove and throw it back. Because the remove node function does not give back the value and that is one of the deficiencies of that function.

So, we will store it inside a temporary integer variable and return it. One way to do this is rewrite the remove node function and return the value of the node as we remove the node. But, we are not going to do that.

STACK FUNCTIONS: peek()

- Peek at the value on the top of the stack
  - Get the value of the node at the front of the linked list
    - Without removing the node

```
int peek(Stack *s){
    return ((s->ll).head)->item;
}
```

 18

Peek function is also a really simple code. All we have to do is to get access to the head pointer. S.ll gets us to the actual struct, not the pointer of the struct. Therefore, using the dot notation, we can access the head pointer and follow it to get the item.

All of these parentheses are not strictly necessary. I will leave it for you to mess around by deleting them and figure out which one is necessary.

Be careful of when you use the dot notation and when you use the arrow notation, depends on whether it's a pointer to a struct or the actual struct itself.

STACK FUNCTIONS: isEmptyStack()

- Check to see if number of nodes == 0
- Make use of the built-in size variable in the LinkedList struct

```
int isEmptyStack(Stack *s){
    if ((s->ll).size == 0) return 1;
    return 0;
}
```

 19

Empty stack is really trivial. Here, we will use the built in size variable and check the value of the size variable.

TODAY

- Motivating application
- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- **Working examples: Applications**

 20

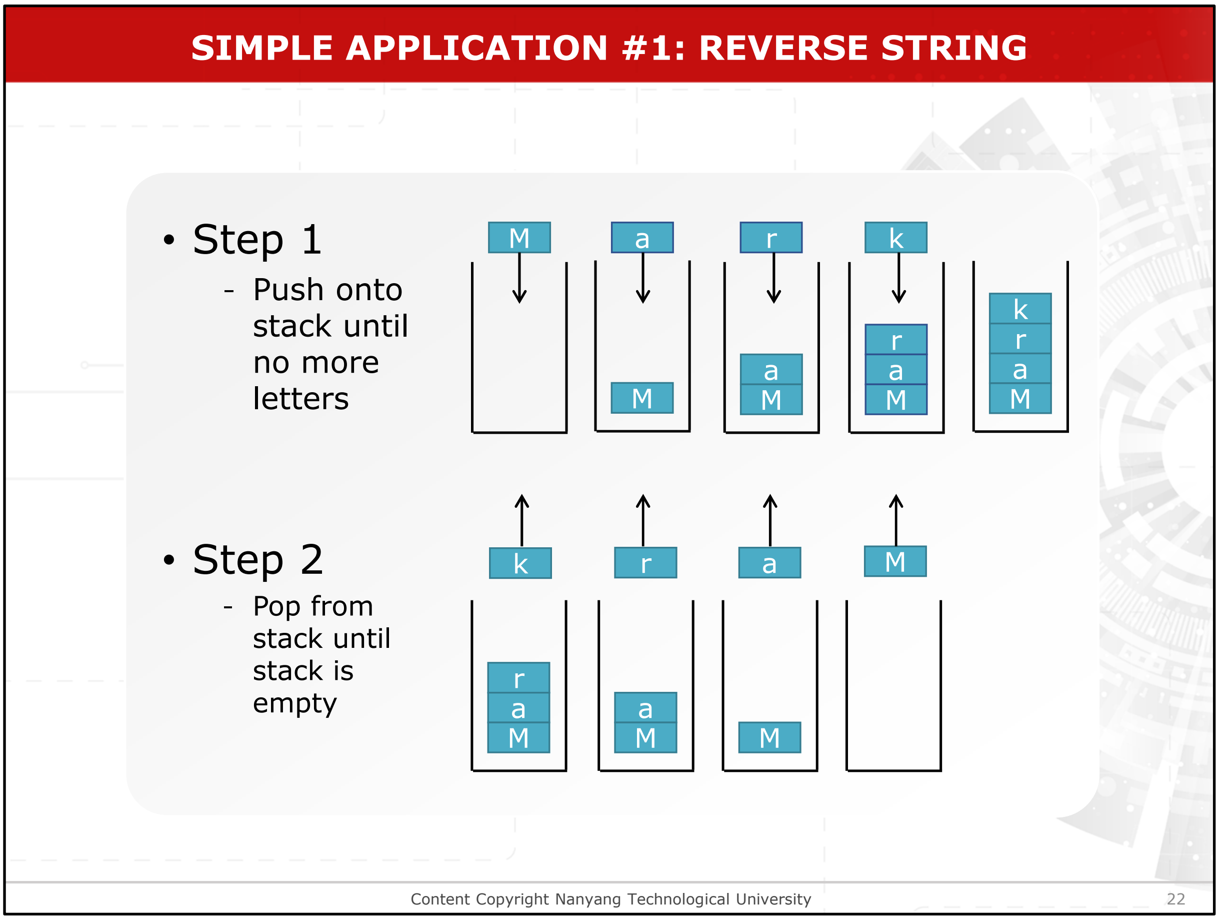
Let me show you this worked example.

SIMPLE APPLICATION #1: REVERSE STRING

- Stacks are useful for reversing items
- Reverse a string: **Mark**
- Idea:
  - Push each letter on the stack
  - When there are no more letters in the original string, pop one by one from the stack
  - The letters will be popped in reverse order from their original position in the string

 21

Stacks are very useful for reversing lists of items. Let's say we have a string and we want to reverse the string. We will use the first in last out property of the stacks. So, we will push one by one to the stack and when we are done, we will pop them off. Likewise, the end letter comes out first, so we will get the string or the list of items in reverse order from the origin order.



Let's say the original string is **M. A. R. K.** In the images you see how the contents build up when we push them into the stack. But, because of the first in last out property, it returns the original string in reverse. Now, why can't we just take the existing character array and start at the end of the string and work our way backwards?

Why can't we write a for loop that goes from n to 0 and we use i--? We cannot do this because you may have to do this procedure not knowing the exact size of the string.

Therefore, what we have to do is keep buffering in all the inputs until the string is done, and then start reversing it.

So, the stack structure allows you to deal with the uncertainty. We do not have to worry about the number of characters up to some extent because the memory blocks are allocated in heap. Therefore, we read them when we need.

At the end of the input, we start to return the input.

SIMPLE APPLICATION #2: REVERSE LIST OF INTEGERS

- Similar to previous application, but with numbers

```
int main(){
    int i = 0;
    Stack s;
    s.ll.head = NULL;

    printf("Enter a number: ");
    scanf("%d", &i);
    while (i != -1){
        push(&s, i);
        printf("Enter a number: ");
        scanf("%d", &i);
    }
    printf("Popping stack: ");
    while (!isEmptyStack(&s))
        printf("%d ", pop(&s));
    return 0;
}
```


The second example will leave to you to look at. It is just like the previous example but with integers this time.

BIGGER APPLICATION: MODELING A CONTAINER YARD

- Scenario:
  - Container yard with 10x10 grid layout, and each square has a stack of containers
  - Overhead crane can move over any stack
  - Crane can hold/inspect one container at a time from top of stack and move it to any other stack
  - Grid square at (0,0) is empty
- Task: Simulate the operation of this container yard

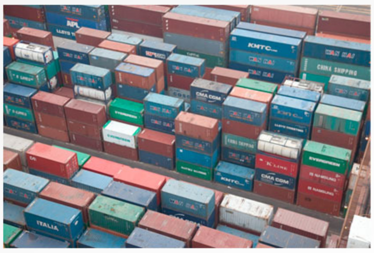


Image credits: http://kaidashton.blogspot.sg/2012/03/hong-kong-container-yards.html, http://es.made-in-china.com/co_goldhorse/product_RTG-Rubber-Tyre-Gantry-Crane-of-Port-for-Container-40FT-or-20FT-Payload-40-Ton_hhruniseg.html

 24

Now, we will discuss about bigger applications such as modelling container yard. How we can use the stack structure to deal with such simulation? How we will map a data structure or different objects with actual items in the yard? And how do we get the 10x10 grid? Because that each of the grid squares contain a stack of containers, we will focus on one operation. We will find a container within a stack.

You will be given a specific row and column of a given stack and you need to find whether the certain container exists inside the stack. Remember, you cannot look at the whole stack and scan all the IDs. You have to pick them one by one and examine the contents and move on.

BIGGER APPLICATION: MODELING A CONTAINER YARD

- Focus on one operation:
  - Find a container within a stack (each has an ID)
  - Given location of the stack (row,col)
  - Given ID of the container

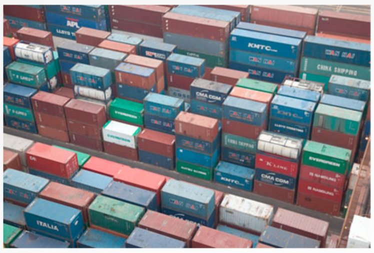


Image credits: http://kaidashton.blogspot.sg/2012/03/hong-kong-container-yards.html, http://es.made-in-china.com/co_goldhorse/product_RTG-Rubber-Tyre-Gantry-Crane-of-Port-for-Container-40FT-or-20FT-Payload-40-Ton_hhruniseg.html

 25

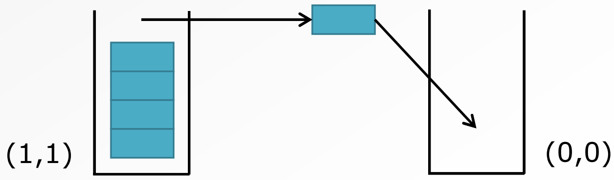
The current stack is limited and only stores integer values. Therefore, we assume that each of the stacks contains the unique integer IDs of the containers.

The basic procedure is that we will pick an item from the stack, check the ID and if it is not we are looking for, we will put it down in somewhere else. Now, where should we put it down? This is where the empty grid square corner comes into use. We have this empty stack at grid 0,0 for all the temporary applications. So, if the picked up item is not what I am looking for, we will put it into the empty stack and look at the next item. We will keep doing this until we find the exact container or we reach the bottom of the stack.

If we reach the bottom, we know that the container that we were looking for is not inside that stack. Therefore, we can return false. Now, regardless of whether we found the container or not, we must replace what we took out from the stack. When you put the items back to the original stack you will see that the order you took them off gets reversed. That means the original stack gets rebuilt in the original order because two first in last out orders cancel each other.

BIGGER APPLICATION: MODELING A CONTAINER YARD

- To find a container with a certain ID (555) in a given stack (1,1):
    - Unload containers one at a time from the top of stack (1,1)
    - If ID matches, report found
    - Else, place container in temporary stack (0,0)
    - Unload next container (repeat)
- When container found or stack empty, need to replace unloaded containers
    - Unload one at a time from temporary stack (0,0)
    - Load one at a time on top of original stack (1,1)



MODELING A CONTAINER YARD (CODE PAGE 1)

```
int main(){

    int i, j, crane, row, col, targetid;
    Stack containeryard[10][10];

    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++){
            containeryard[i][j].ll.head = NULL;
            containeryard[i][j].ll.size = 0;
        }

    row = col = 1;
    targetid = 555;

    // Initialize the target stack at (1,1)
    // Each container has an ID number - this will go in the stack
    for (i = 0; i < 10; i++)
        push(&(containeryard[row][col]), i*100+i*10+i);
```

 27

If you look at the code, you will see we have stack containeryard. This is an array of stacks. You have previously dealt with two dimensional array of integers and array of structs. Now here you have two dimensional array of structs.

How do we access the individual containers? We need to go to the specific stack inside the two dimensional array and push and pop from these stacks. The nested for loops are for initialization. So all I'm doing is setting everything to zero by setting the size to zero, setting the head to null, because I need this to initialize all the linked list inside my stack.

Unfortunately in C. there's no way to automatically initialize my linked list structure when I first create it, so I need to initialize it separately. Now, we are looking for a specific ID. For that we need to create a fake stack of containers. Therefore, we push all the numbers in to the stack so we will have 111, 222, 333, 444, 555, and so on that represent the containers.

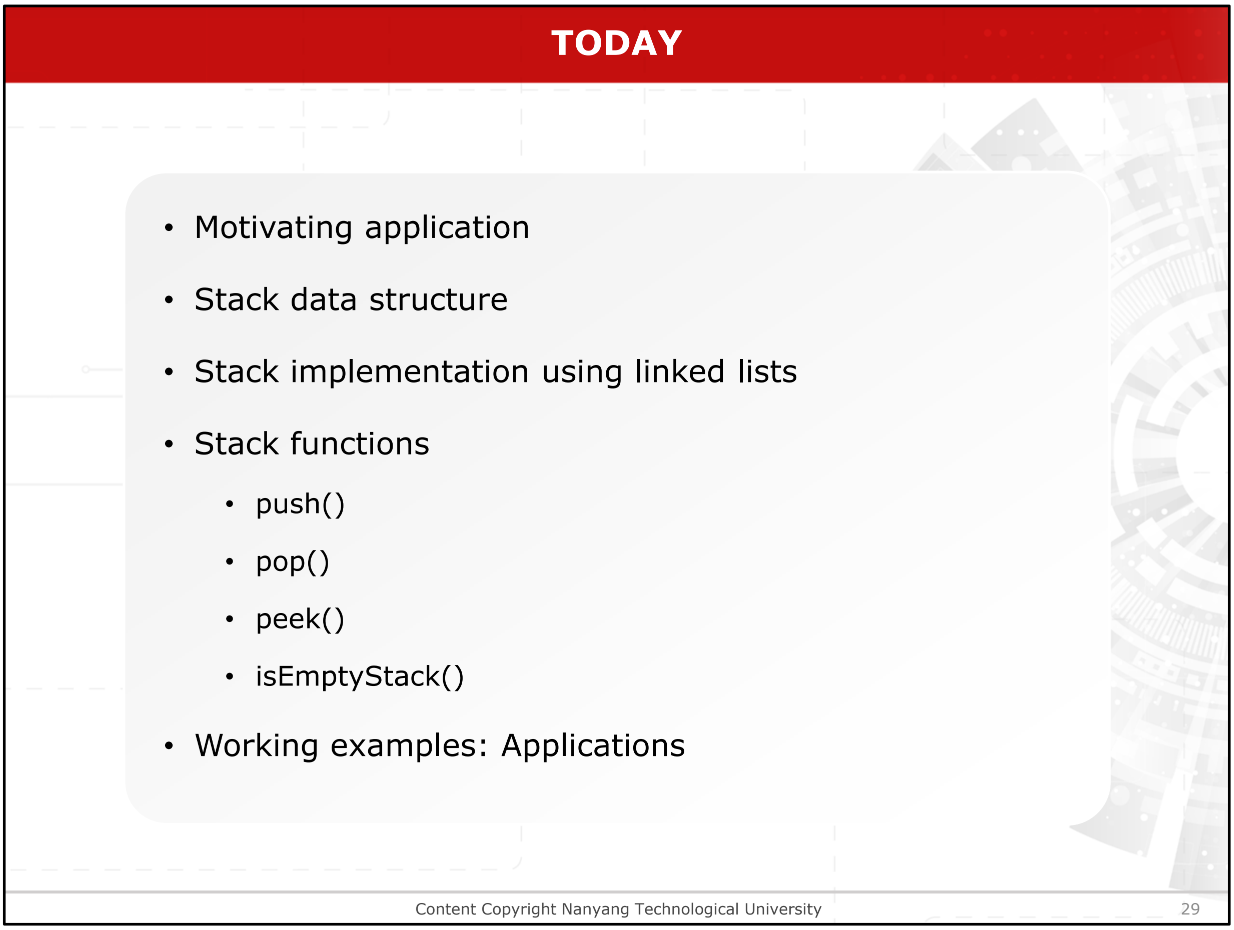
MODELING A CONTAINER YARD (CODE PAGE 2)

```
    // Find a container within a stack
    // Row, col of stack and ID of container are given
    while (!isEmptyStack(&(containeryard[row][col]))){
        crane = pop(&(containeryard[row][col]));

        // Container found
        if (crane == targetid)
            break;

        // Still not found, so store this crane temporarily in (0,0)
        push(&(containeryard[0][0]), crane);
    }

    // Need to rebuild the original stack
    while (!isEmptyStack(&(containeryard[0][0])))
        push(&(containeryard[row][col]),
pop(&(containeryard[0][0])));

    if (crane == targetid)
        printf("Container found!\n");

    return 0;
}
```

 28

Now we start the process that we learned earlier. We pick up the top container, check the ID and if it is the one we want, we return the true value to the calling function and replace everything.

If we did not find the container, we will reach the bottom and return false and put everything back in place.



Today we learned about stacks and its functions. The next day we'll talk about the queue data structures.