Lecture 5_Data Structure

Queues

1. *Queues* are one form of abstract data structure.
2. Properties:*FIFO* "first in first out" (This is where the name queues come from)
3. Actions:
    1. Enqueued: an item joins the line or queue
    2. Dequeued: an item leaves the queue once it reaches the front of the line

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
}
queue;
```

```
where person is another struct and size represents the actually size of the queue, regardless of its capacity.
```

Stacks

1. *Stacks* are one form of abstract data structure.
2. Properties:*FIFO* "first in last out"(like stacking trays, the last tray pushed to the stack leaves the first)
3. Actions:
    1. Push: place something on top of a stack.
    2. Pop: remove something from the top of the stack.

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
}
stack;
```

```
Actually same as queues
```

4. Limitation for both stacks and queues: Since the capacity of the array is always predetermined in the code. Therefore, the stack may always be oversized.

Resizing Arrays

1. An array is a block of contiguous memory.
2. If we have an array of three integers and hope to add one more int, one way to achieve this is to reallocate memories of 4 *sizeof(int) then copy the data from the initial array. One of the drawbacks of this approach is that it's bad design: Every time we add a number, we have to copy the array item by item.

```c
// Implements a list of numbers with an array of dynamic size

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // List of size 3
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Initialize list of size 3 with numbers
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // List of size 4
    int *tmp = malloc(4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
        return 1;
    }

    // Copy list of size 3 into list of size 4
    for (int i = 0; i < 3; i++)
    {
        tmp[i] = list[i];
    }
```

```
    // Add number to list of size 4
    tmp[3] = 4;

    // Free list of size 3
    free(list);

    // Remember list of size 4
    list = tmp;

    // Print list
    for (int i = 0; i < 4; i++)
    {
        printf("%i\n", list[i]);
    }

    // Free list
    free(list);
    return 0;
}
```

We can avoid using for loop by using realloc

```
// Implements a list of numbers with an array of dynamic size using realloc

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // List of size 3
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Initialize list of size 3 with numbers
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // Resize list to be of size 4
    int *tmp = realloc(list, 4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
```

```
        return 1;
    }
    list = tmp;

    // Add number to list
    list[3] = 4;

    // Print list
    for (int i = 0; i < 4; i++)
    {
        printf("%i\n", list[i]);
    }

    // Free list
    free(list);
    return 0;
}
```

`realloc`

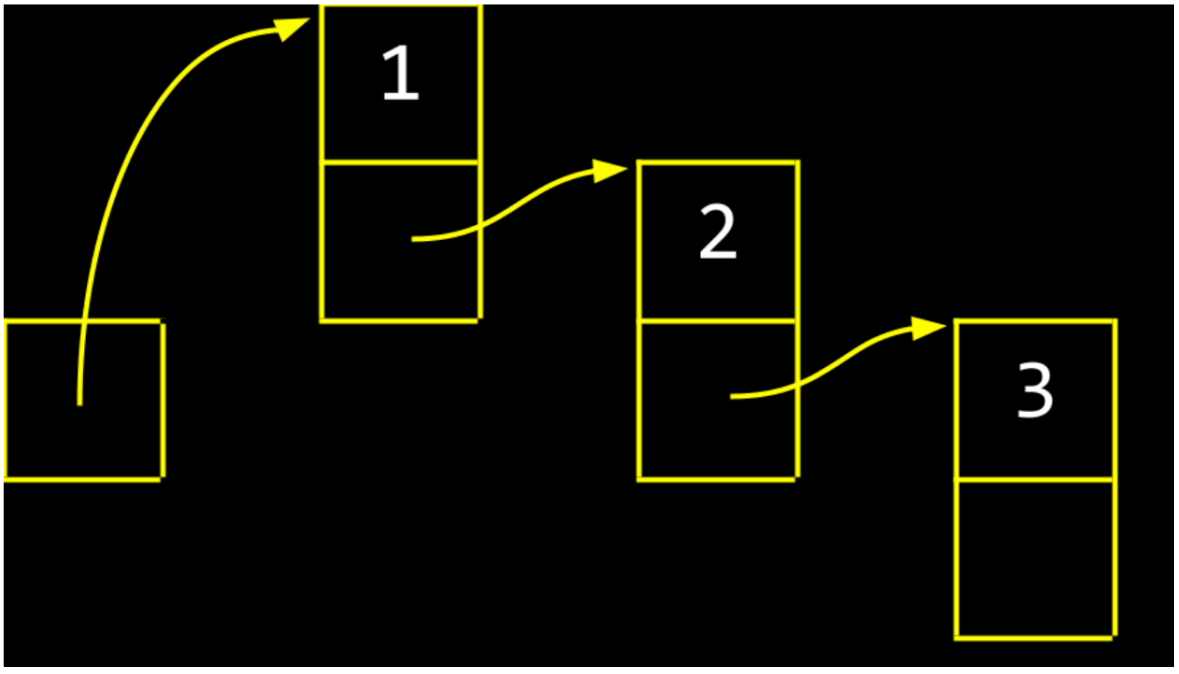
1. If there is enough memory blocks following the initial memory blocks, realloc function would directly append the blocks needed
2. If the size of available blocks is not enough, the function would first allocate the blocks needed in a new space, then copy from the original array item by item, and automatically free the initial memory block
   One reminder is that it is bad practice to allocate way more memory needed for the list.

Linked Lists

1. Important operator: `->` This operator goes to an address and looks inside a structure.
2. Linked lists allow us to stitch values in varying areas of memory together, that is to dynamically grow and shrink the list as we desire.
3. The way we achieve this is by using more memories as pointers, pointing to the address of next value.

1. we would keep one more element in memory, a pointer, that keeps track of the first item in the list, called the *head* of the list.
2. NULL is utilized to indicate that nothing else is *next* in the list.

Abstracting away the memory addresses, the list would appear as follows:



4. These boxes are called *nodes*. A *node* contains both an *item* and a pointer called *next*. Note that every node in the linked lists must be a pointer,

```
typedef struct node
{
    int number;
    struct node *next;
    // here we use struct node to clarify the next points to a node since
node hasn't been created yet
```

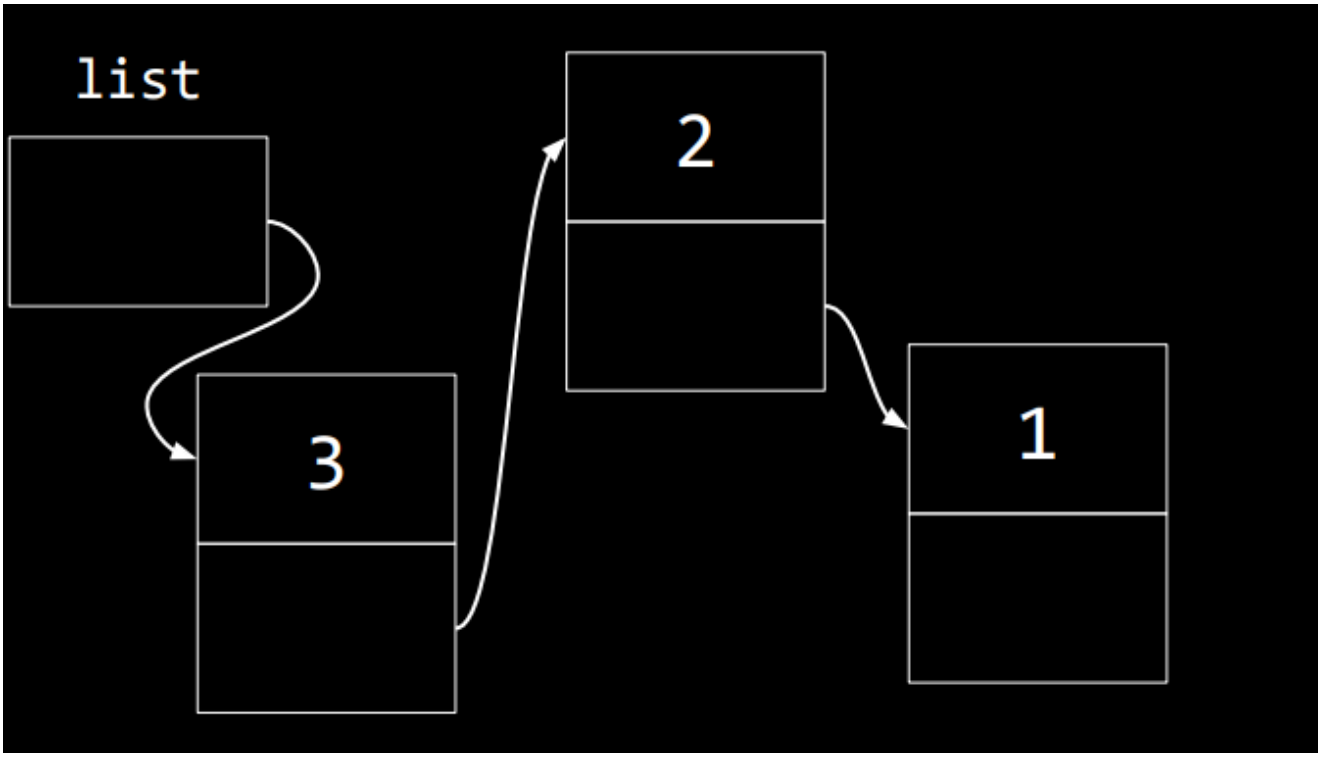
```
}
node;
```

5. One completion of a linked list (prepend):

```c
// Start to build a linked list by prepending nodes

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int number;
    struct node *next;
} node;

int main(void)
{
    // Memory for numbers
    node *list = NULL;// the list is the head of the linked list

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));//create nodes
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");// fill the nodes with the number
        // Prepend node to list
        n->next = list;
        // points to the address of the previous node
        // for the first node created, that would be null
        list = n;// the head points to the newest node
    }
    return 0;
}
```

```
Note that in line 22, each node is created as a pointer
Actually,every node in the linked list should be a pointer because when we use malloc to allocate memory blocks, it returns a pointer to the address of the first memory block, and we need a pointer to receive this pointer.
```

In the above code, the linked list is actually created in a reverse way: the last number added is the first number that appears in the list. Accordingly, if we print the list in order, starting with the first node, the list will appear out of order.



We can add the following code to print the list in the order of the pointers but the reverse way of the use input(i.e. 3 2 1)

```
    // Print numbers
    node *ptr = list;
    // create a new pointer that points to what list is pointing to
    // now it is pointing to the head of the linked list
    while (ptr != NULL)// only the next pointer of the last node is null
    {
        printf("%i\n", ptr->number);
        ptr = ptr->next;
    }
    return 0;
}
```

- Inserting into the list is always in the order of O(1), as it only takes a very small number of steps to insert at the front of a list.
- Time required to search this list is in the order of O(n), because in the worst case the entire list must always be searched to find an item.

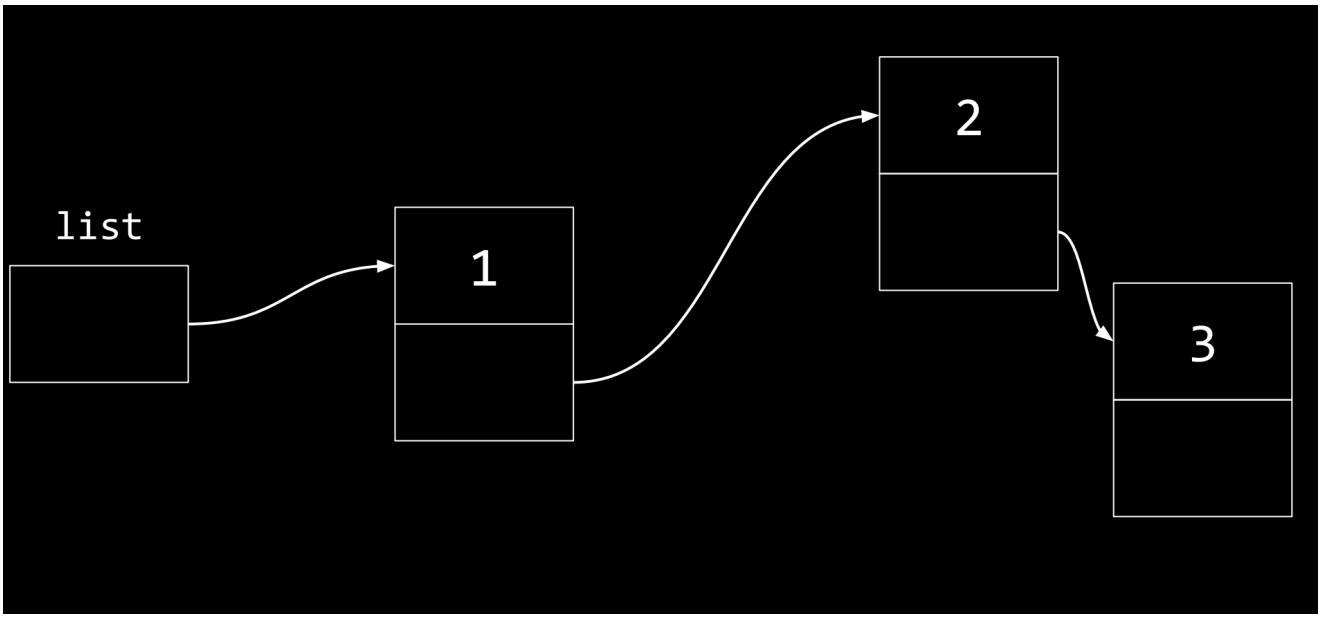
6. Linked lists are not stored in a contiguous block of memory. They can grow as large as you wish, provided that enough system resources exist. The downside is that more memory is required to keep track of the list instead of an array. For each element you

must store not just the value of the element, but also a pointer to the next node. Further, linked lists cannot be indexed like an array because we need to pass through the first n−1 elements to find the location of the nth element. Because of this, the list pictured above must be linearly searched. Binary search, therefore, is not possible in a list constructed as above.

7. Another completion of a linked list (append):

```c
// Appends numbers to a link list

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int number;
    struct node *next;
} node;

int main(void)
{
    // Memory for numbers
    node *list = NULL;
    // first create a pointer called list and clear its garbage value

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));// create a node
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");
        n->next = NULL;//clear its garbage value

        // If list is empty
        if (list == NULL)
        {
            // This node is the whole list
            list = n;
        }
        // in the first iteration, list should point to n
        // different from prepending, the list always points to the head
        // because now n is the first node
```

```
        // If list has numbers already
        else
        {
            // Iterate over nodes in list
            for (node *ptr = list; ptr != NULL; ptr = ptr->next)
            // create another pointer that points to where list points at
            // when it is not NULL means the for loop ends
            // after ptr points to the last node
            {
                // If at end of list
                if (ptr->next == NULL)
                // when pointer points to the last node
                {
                    // Append node
                    ptr->next = n;
                    // the last node points to the newly created node
                    break;
                    // break out of the for loop and go creating new nodes
                }
            }
        }
    }

    // Print numbers
    for (node *ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%i\n", ptr->number);
    }

    // Free memory
    node *ptr = list;
    // a pointer ptr points to the head
    while (ptr != NULL)
    // stop after pointing at the last node
    {
        node *next = ptr->next;
        // create another pointer next that points to the next node
        free(ptr);
        // free the current node
        ptr = next;
        // points to the next node
    }
    return 0;
    // indicating success
}
```



Running time

- When appending an element (adding to the end of the list) our code will run in $O(n)$, as we have to go through our entire list before we can add the final element

8. Another implementation to sort the linked list:

```c
// Implements a sorted linked list of numbers

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int number;
    struct node *next;
} node;

int main(void)
{
    // Memory for numbers
    node *list = NULL;

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            return 1;
        }
```

```
        n->number = get_int("Number: ");
        n->next = NULL;

        // If list is empty
        if (list == NULL)
        {
            list = n;
        }
    // The above code is the same
    // we discuss the three possibilities of the position of new node

        // If number belongs at beginning of list
        else if (n->number < list->number)
        // if the new node is smaller than the head of the linked list
        // it should be prepended
        {
            n->next = list;
            // list points to the initial head
            // which becomes the second node
            list = n;
            // n is now the head
        }

        // If number belongs later in list
        else
        {
            // Iterate over nodes in list
            for (node *ptr = list; ptr != NULL; ptr = ptr->next)
            {
                // If at end of list
                if (ptr->next == NULL)
                {
                    // Append node
                    ptr->next = n;
                    break;
                }

                // If in middle of list
                if (n->number < ptr->next->number)
                // number of the new node is smaller than the next node
                // it should be added after the current node
                {
                    n->next = ptr->next;
                    ptr->next = n;
                    break;
                }
            }
        }
    }
```

```
    // below is the same
    // Print numbers
    for (node *ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%i\n", ptr->number);
    }

    // Free memory
    node *ptr = list;
    while (ptr != NULL)
    {
        node *next = ptr->next;
        free(ptr);
        ptr = next;
    }
    return 0;
}
```

Running time

- To insert an element in this specific order, our code will still run in O(n) for each insertion, as in the worst case we will have to look through all current elements.

Trees 考完220再来写

1.