Com S // Cpr E // Math 5250

Numerical Analysis of High-Performance Computing

Instructor: Songting Luo

Lecture 12: Stacks & Queues and Best Practices in Scientific Computing

Outline

1. Review of Linked Lists
2. Stacks & Queues
3. Best practices in Scientific Computing

Review of Linked Lists

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/node.h:

```
1  #ifndef __NODE_H__
2  #define __NODE_H__
3
4  typedef struct node node;
5  struct node
6  {
7     int position;
8     int value;
9     node* next;
10 };
11
12 // Functions associated with struct node
13 int GetNumberOfNodes();
14 void GenerateList(node** head,const int num);
15 void Print(const int forward,const node* head);
16 void PrintList(const node* head);
17 void ReversePrintList(const node* head);
18 int GetKey();
19 void SearchList(const node* head,const int key);
20 void DeleteList(node** head);
21
22 #endif
```

Linked Lists

$ISUHPC/lectures/lecture11/codes/linked_list/main.c:

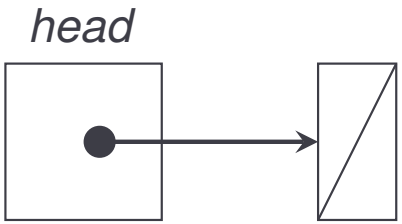
```
1  int main()
2  {
3     // Declare the head node in my list
4     node* head = NULL;
5
6     // Set number of nodes and generate a new list
7     const int num_nodes = GetNumberOfNodes();
8     GenerateList(&head,num_nodes);
9
10    // Print list to screen
11    Print(1,head); // foward print
12    Print(0,head); // reverse print
13
14    // Ask for a key, then search list
15    if(num_nodes>0)
16    {
17       const int key = GetKey();
18       SearchList(head,key);
19    }
20
21    // Delete list (free up memory)
22    DeleteList(&head);
23 }
```

Linked Lists

Example: Add 3 nodes to the initially empty list. First node has a value of **3**, second node has a value of **11**, and third node has a value of **7**.

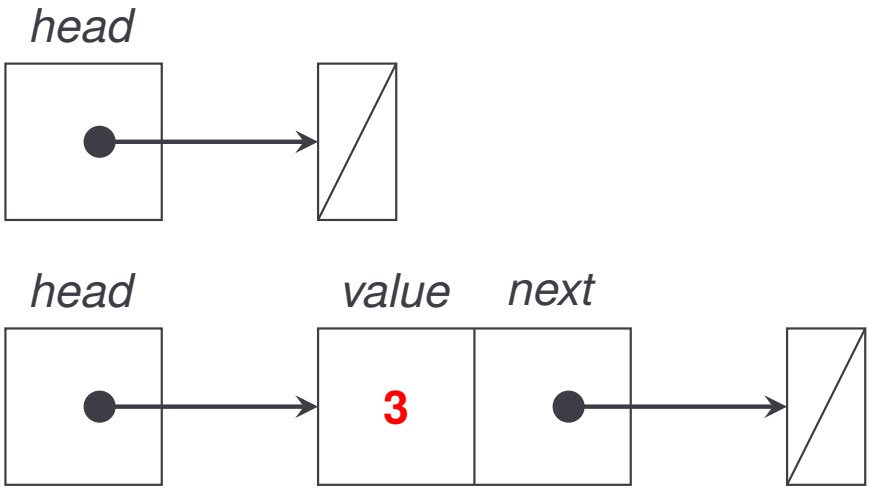
Linked Lists

Example: Add 3 nodes to the initially empty list. First node has a value of **3**, second node has a value of **11**, and third node has a value of **7**.



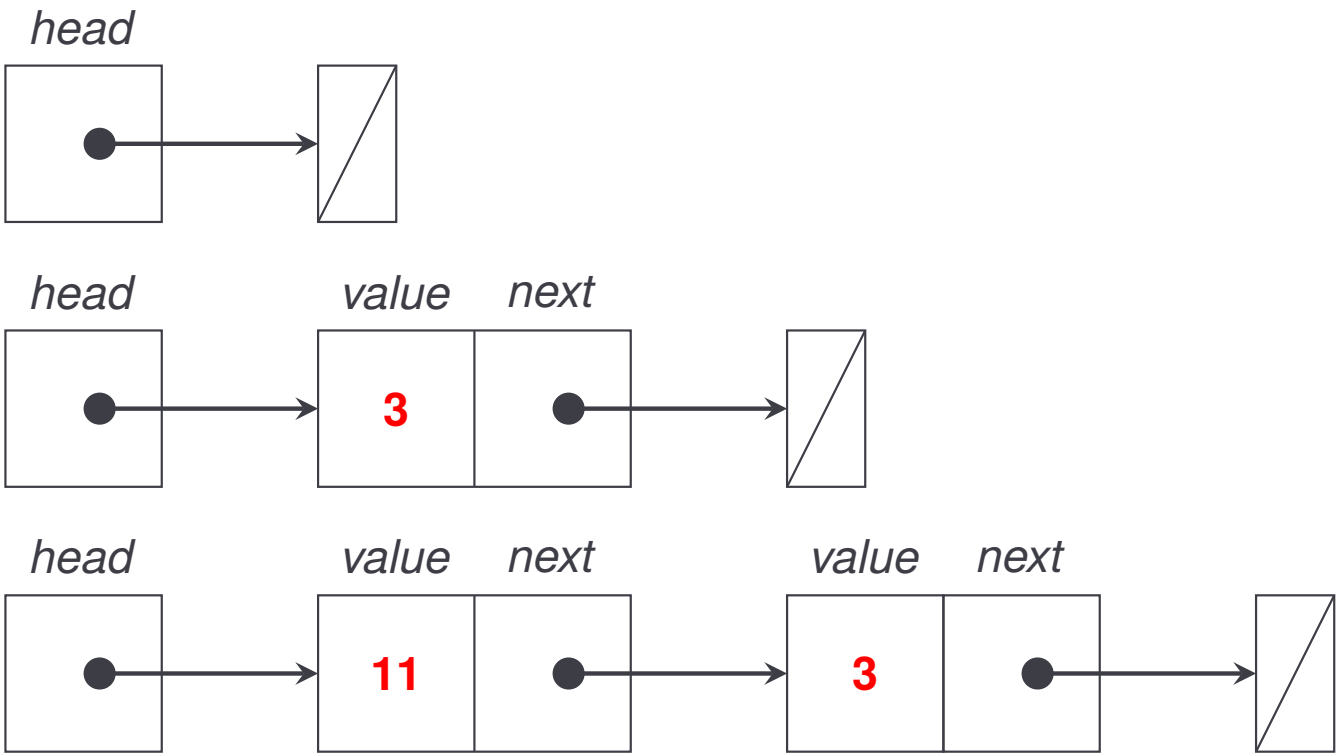
Linked Lists

Example: Add 3 nodes to the initially empty list. First node has a value of **3**, second node has a value of **11**, and third node has a value of **7**.



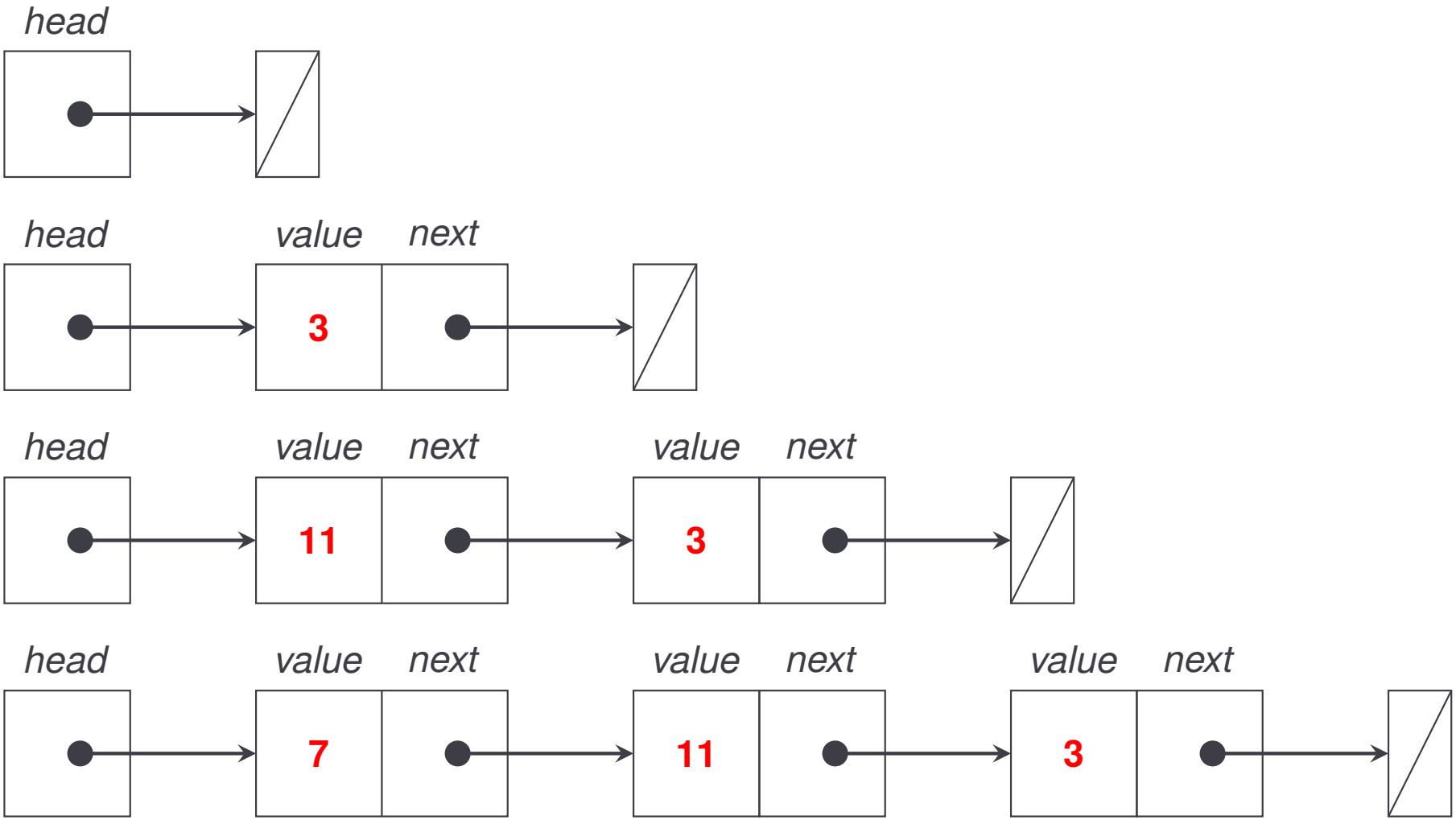
Linked Lists

Example: Add 3 nodes to the initially empty list. First node has a value of **3**, second node has a value of **11**, and third node has a value of **7**.



Linked Lists

Example: Add 3 nodes to the initially empty list. First node has a value of **3**, second node has a value of **11**, and third node has a value of **7**.



Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/GenerateList.c:

```
1  void GenerateList(node** head, const int num_nodes)
2  {
3     node* temp; srand( time(NULL) );
4
5     for (int i=0; i<num_nodes; i++)
6     {
7        temp = (node*)malloc(sizeof(node));
8        temp->value = rand()%num_nodes; temp->position = 0;
9        printf("%4i",temp->value);
10
11       if (*head == NULL)
12       {
13          *head = temp;
14          (*head)->next = NULL;
15       }
16       else
17       {
18          temp->next = *head;
19          *head = temp;
20       }
21    }
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/GenerateList.c:

```
    printf("\n");

    node* ptr = *head; int pos = 1;
    while(ptr!=NULL)
    {
        ptr->position = pos;
        pos = pos+1;
        ptr = ptr->next;
    }
}
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/Print.c:

```
1  void Print(const int forward, const node* head)
2  {
3     if (head==NULL)
4     { printf(" List is emtpy.\n\n"); return; }
5     printf("\n");
6     printf(" ---------------------------------------------\n");
7     printf(" |Pos:|Val:|    Address:    |      Next:     |\n");
8     printf(" ---------------------------------------------\n");
9     switch(forward)
10    {
11       case 0:
12          ReversePrintList(head);
13          break;
14       case 1:
15          PrintList(head);
16          break;
17       default:
18          printf("\n Error: forward must be 0 or 1.\n");
19          printf("  forward = %i\n",forward); exit(1);
20    }
21    printf(" ---------------------------------------------\n");
22 }
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/Print.c:

```
1  void PrintLine(const int pos,const int val,
2                 const node* head,const node* next)
3  { printf(" |%3i |%3i |%15p |%15p |\n",pos,val,head,next); }
4
5  void PrintList(const node* head)
6  {
7     PrintLine(head->position,head->value,head,head->next);
8     if (head->next == NULL)
9     { return; }
10    PrintList(head->next);
11 }
12
13 void ReversePrintList(const node* head)
14 {
15    if (head->next == NULL)
16    {
17       PrintLine(head->position,head->value,head,head->next);
18       return;
19    }
20    ReversePrintList(head->next);
21    PrintLine(head->position,head->value,head,head->next);
22 }
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/GetKey.c:

```c
#include <stdio.h>

int GetKey()
{
    int key;
    printf("\n Enter key to search: ");
    scanf("%i", &key);

    return key;
}
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/SearchList.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "node.h"
4
5  void SearchList(const node* head,
6                  const int key)
7
8  {
9     if (head->value==key)
10    {
11       printf(" Key found at Position: %i\n",
12              head->position);
13    }
14
15    if (head->next==NULL)
16    {
17       printf("\n"); return;
18    }
19
20    SearchList(head->next,key);
21 }
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/DeleteList.c:

```
1  #include <stdlib.h>
2  #include "node.h"
3
4  void DeleteList(node** head)
5  {
6     node* temp;
7     while (*head!=NULL)
8     {
9        temp = *head;
10       *head = (*head)->next;
11       free(temp);
12    }
13 }
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/Makefile:

```
1  # $ISUHPC/lectures/lecture11/codes/linked_list/Makefile
2  CC = gcc
3  FFLAGS = -O3 -Wall
4  LFFLAG =
5  OBJECTS = main.o \
6  GetNumberOfNodes.o \
7  GenerateList.o \
8  Print.o \
9  GetKey.o \
10 SearchList.o \
11 DeleteList.o
12
13 .PHONY: clean help
14
15 main.exe: $(OBJECTS)
16         $(CC) $(LFLAGS) $(OBJECTS) -o main.exe
17
18 %.o: %.c
19         $(CC) $(FFLAGS) -c $<
20
21 clean:
22         rm -f $(OBJECTS) main.exe
```

Linked lists

$ISUHPC/lectures/lecture11/codes/linked_list/Makefile:

```
help:
        @echo "Valid targets:"
        @echo "  main.exe"
        @echo "  main.o"
        @echo "  GetNumberOfNodes.o"
        @echo "  GenerateList.o"
        @echo "  Print.o"
        @echo "  GetKey.o"
        @echo "  SearchList.o"
        @echo "  DeleteList.o"
        @echo "  clean:  removes *.o and *.exe files"
```

```
$ make
gcc -O3 -Wall -c main.c
gcc -O3 -Wall -c GetNumberOfNodes.c
gcc -O3 -Wall -c GenerateList.c
gcc -O3 -Wall -c Print.c
gcc -O3 -Wall -c GetKey.c
gcc -O3 -Wall -c SearchList.c
gcc -O3 -Wall -c DeleteList.c
gcc  main.o GetNumberOfNodes.o GenerateList.o Print.o GetKey.o
SearchList.o DeleteList.o -o main.exe
```

Linked lists

```
 $ main.exe

  Enter the number of nodes: 10

  0   9   4   9   1   3   5   9   0   2

-------------------------------------------
|Pos:|Val:|    Address:    |      Next:     |
-------------------------------------------
|  1 |  2 | 0x7ffe20404d30 | 0x7ffe20404d20 |
|  2 |  0 | 0x7ffe20404d20 | 0x7ffe20404d10 |
|  3 |  9 | 0x7ffe20404d10 | 0x7ffe20404d00 |
|  4 |  5 | 0x7ffe20404d00 | 0x7ffe20404cf0 |
|  5 |  3 | 0x7ffe20404cf0 | 0x7ffe20404ce0 |
|  6 |  1 | 0x7ffe20404ce0 | 0x7ffe20404cd0 |
|  7 |  9 | 0x7ffe20404cd0 | 0x7ffe20404cc0 |
|  8 |  4 | 0x7ffe20404cc0 | 0x7ffe20404cb0 |
|  9 |  9 | 0x7ffe20404cb0 | 0x7ffe20404ca0 |
| 10 |  0 | 0x7ffe20404ca0 |            0x0 |
-------------------------------------------
```

Linked lists

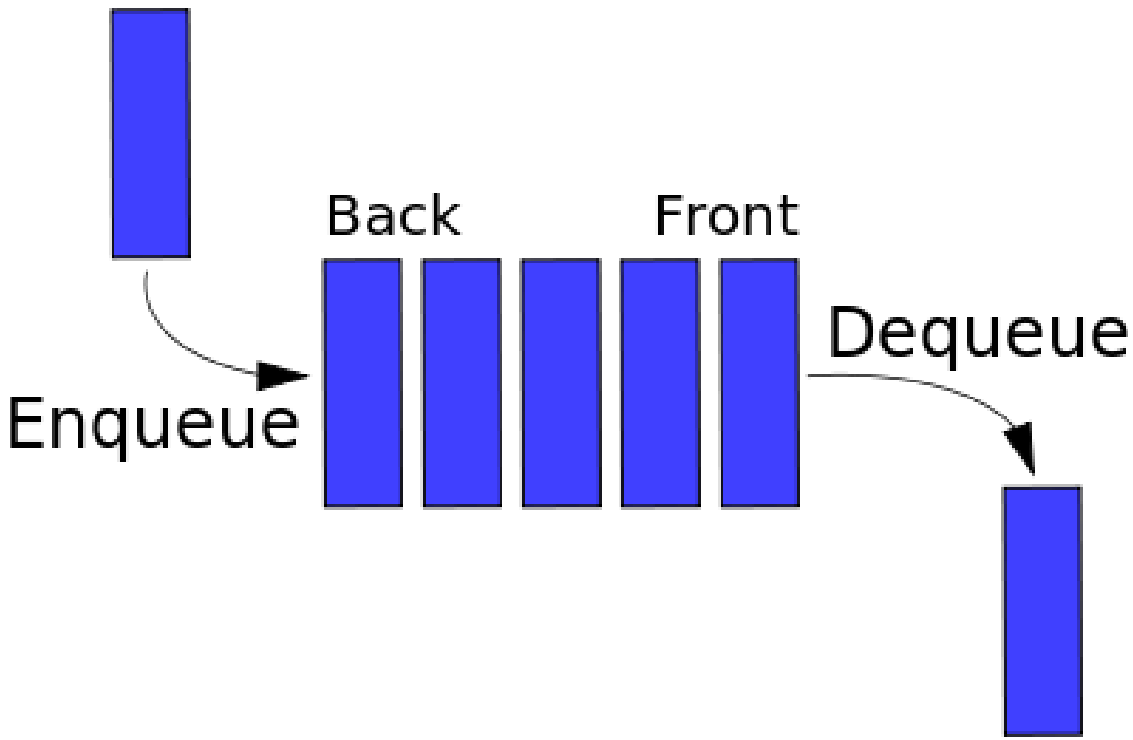
```
-------------------------------------------
|Pos:|Val:|    Address:    |      Next:     |
-------------------------------------------
| 10 |  0 | 0x7ffe20404ca0 |            0x0 |
|  9 |  9 | 0x7ffe20404cb0 | 0x7ffe20404ca0 |
|  8 |  4 | 0x7ffe20404cc0 | 0x7ffe20404cb0 |
|  7 |  9 | 0x7ffe20404cd0 | 0x7ffe20404cc0 |
|  6 |  1 | 0x7ffe20404ce0 | 0x7ffe20404cd0 |
|  5 |  3 | 0x7ffe20404cf0 | 0x7ffe20404ce0 |
|  4 |  5 | 0x7ffe20404d00 | 0x7ffe20404cf0 |
|  3 |  9 | 0x7ffe20404d10 | 0x7ffe20404d00 |
|  2 |  0 | 0x7ffe20404d20 | 0x7ffe20404d10 |
|  1 |  2 | 0x7ffe20404d30 | 0x7ffe20404d20 |
-------------------------------------------

Enter key to search: 9
Key found at Position: 3
Key found at Position: 7
Key found at Position: 9
```

Stacks & Queues

Queues

Queue

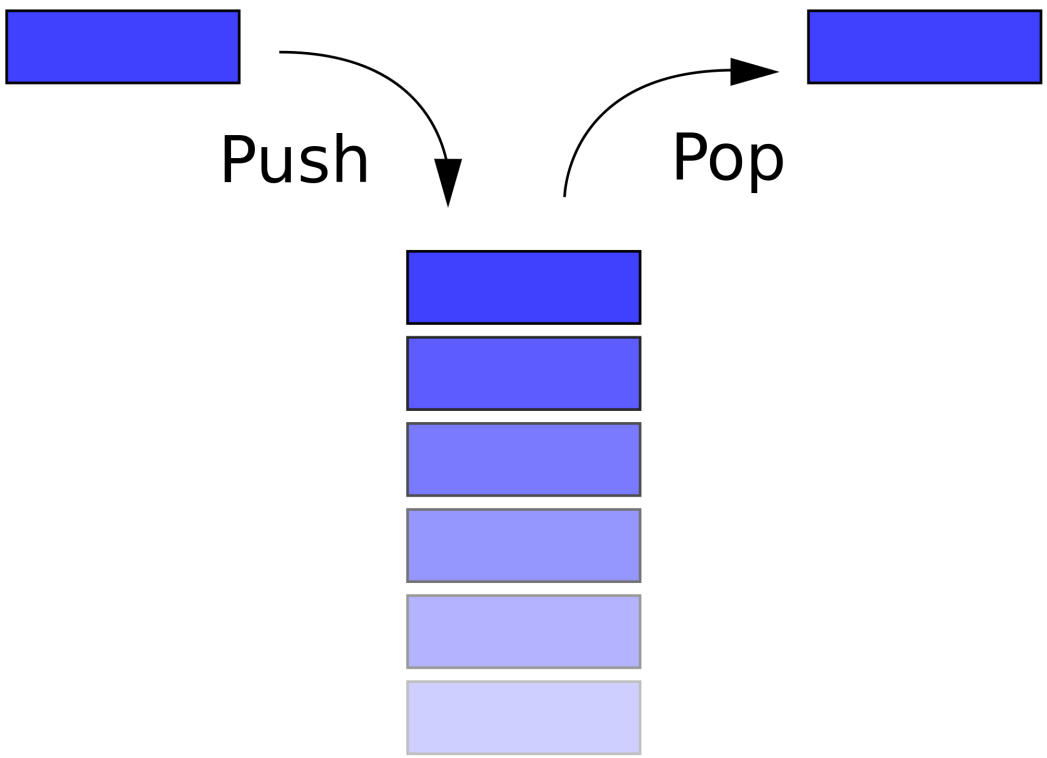
A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: **push** (or **enqueue**), which adds an element to the end of the queue, and **pop** (or **dequeue**), which removes the first element at the front of the queue.



Stacks

Stack

A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: **push**, which adds an element to the collection, and **pop**, which removes the last element that was added.



Stacks

$ISUHPC/lectures/lecture12/codes/stack/main.c:

```
1  #include <stdlib.h>
2  #include "node.h"
3
4  int main()
5  {
6     node* top = NULL;
7
8     int option=0;
9     while(option!=6)
10    {
11       ExecuteOption(option,&top);
12       option = QueryOption();
13    }
14
15    DeleteStack(&top);
16    return 0;
17 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/DisplayOptions.c:

```c
#include <stdio.h>

void DisplayOptions()
{
    printf("\n 0 - List Options");
    printf("\n 1 - Push (a single node)");
    printf("\n 2 - Pop (the top node)");
    printf("\n 3 - Peek (at the top node)");
    printf("\n 4 - Dipslay (the entire stack)");
    printf("\n 5 - Get stack size");
    printf("\n 6 - Exit");
    printf("\n");
}
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Option.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "node.h"
4
5  int QueryOption()
6  {
7     int option;
8     printf(" ENTER CHOICE : ");
9     scanf("%i", &option);
10    return option;
11 }
12
13 void ExecuteOption(const int option, node** top)
14 {
15    int value;
16    switch(option)
17    {
18       case 0: // Display available options
19          DisplayOptions();
20          break;
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Option.c:

```
        case 1: // Enter a new value then push new node to stack
           printf(" Enter value : ");
           scanf("%i", &value);
           Push(value,top);
           break;
        case 2: // Pop top value off of stack
           if (*top!=NULL)
           {
              Pop(top,&value);
              printf(" Pop value = %i\n",value);
           }
           else
           { printf(" Stack is empty.\n"); }
           break;
        case 3: // Peek at top value on stack
           if((*top)!=NULL)
           {
              value = Peek(*top);
              printf(" Top value is %i\n",value);
           }
           else
           { printf(" Stack is empty.\n"); }
           break;
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Option.c:

```
        case 4: // Display the entire stack
           DisplayStack(*top);
           break;
        case 5: // Print stack size
           GetStackSize(*top,&value);
           printf(" Stack size is %i\n",value);
           break;
        case 6: // Do nothing here, but this causes code to end
           break;
        default:
           printf("Error: incorrect option. Try again.\n");
           break;
    }
 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Push.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "node.h"
4
5  void Push(const int input, node** top)
6  {
7     if (*top == NULL)
8     {
9        *top =(node*)malloc(sizeof(struct node));
10       (*top)->next = NULL;
11       (*top)->value = input;
12       (*top)->position = 1;
13    }
14    else
15    {
16       node* temp;
17       temp =(node*)malloc(sizeof(struct node));
18       temp->next = *top;
19       temp->value = input;
20       temp->position = 1;
21       *top = temp;
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Push.c:

```
1
2           node* ptr = (*top)->next;
3           while (ptr!=NULL)
4           {
5              ptr->position = ptr->position+1;
6              ptr = ptr->next;
7           }
8       }
9   }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Pop.c:

```
1  void Pop(node** top, int* output)
2  {
3     node* temp = *top;
4
5     if (temp==NULL)
6     { return; }
7     else
8     { temp = temp->next; }
9     *output = (*top)->value;
10    free(*top);
11    *top = temp;
12
13    node* ptr = *top;
14    while (ptr!=NULL)
15    {
16       ptr->position = ptr->position-1;
17       ptr = ptr->next;
18    }
19 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Peek.c:

```
1  #include "node.h"
2
3  int Peek(node* top)
4  {
5      return top->value;
6  }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/DisplayStack.c:

```
1  void DisplayStack(node* top)
2  {
3     if (top==NULL)
4     { printf(" Stack is emtpy.\n"); return; }
5
6     printf(" ------------------------------------------\n");
7     printf(" |Pos:|Val:|    Address:    |      Next:     |\n");
8     printf(" ------------------------------------------\n");
9     PrintNode(top);
10    printf(" ------------------------------------------\n");
11 }
12
13 void PrintNode(node* top)
14 {
15    printf(" |%3i |%3i |%15p |%15p |\n",
16           top->position,top->value,top,top->next);
17    if (top->next == NULL)
18    { return; }
19    PrintNode(top->next);
20 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/GetStackSize.c:

```
1  #include <stdlib.h>
2  #include "node.h"
3
4  void GetStackSize(node* top, int* stack_size)
5  {
6     if (top==NULL)
7     { *stack_size = 0; return; }
8
9     if (top->next == NULL)
10    { *stack_size = top->position; return; }
11    GetStackSize(top->next,stack_size);
12 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/DeleteStack.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "node.h"
4
5  void DeleteStack(node** top)
6  {
7     node* temp;
8     while (*top!=NULL)
9     {
10       temp = *top;
11       *top = (*top)->next;
12       free(temp);
13     }
14
15    printf("\n Goodbye!\n\n");
16 }
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Makefile:

```
# $ISUHPC/lectures/lecture12/codes/stack/Makefile
CC = gcc
FFLAGS = -O3 -Wall
LFFLAG =
OBJECTS = main.o \
Option.o \
DisplayOptions.o \
Push.o \
Pop.o \
Peek.o \
DisplayStack.o \
GetStackSize.o \
DeleteStack.o

.PHONY: clean help

main.exe: $(OBJECTS)
        $(CC) $(LFLAGS) $(OBJECTS) -o main.exe

%.o: %.c
        $(CC) $(FFLAGS) -c $<
```

Stacks

$ISUHPC/lectures/lecture12/codes/stack/Makefile:

```
1 clean:
2         rm -f $(OBJECTS) main.exe
```

```
$ make
gcc -O3 -Wall -c main.c
gcc -O3 -Wall -c Option.c
gcc -O3 -Wall -c DisplayOptions.c
gcc -O3 -Wall -c Push.c
gcc -O3 -Wall -c Pop.c
gcc -O3 -Wall -c Peek.c
gcc -O3 -Wall -c DisplayStack.c
gcc -O3 -Wall -c GetStackSize.c
gcc -O3 -Wall -c DeleteStack.c
gcc  main.o Option.o DisplayOptions.o Push.o Pop.o
Peek.o DisplayStack.o GetStackSize.o DeleteStack.o
-o main.exe
```

Best practices in Scientific Computing

Best practices

Reference:

G. Wilson, D.A. Aruliah, C.T. Brown, et al., *Best Practices in Scientific Computing*, `http://arxiv.org/abs/1210.0530`.

1. Write programs for people, not computers

Instead of

```
def rectangle_area(x1,y1,x2,y2):
    ... do something ...
```

Write

```
def rectangle_area(point1,point2):
    ... do something ...
```

Corollary: Give variable meaningful names that make the code easier to understand

Best practices

2. Let the computer handle repetitive tasks

> *Science often involves repetition of computational tasks such as processing large numbers of data files in the same way or regenerating figures each time new data is added to an existing analysis. Computers were invented to do these kinds of repetitive tasks but, even today, many scientists type the same commands in over and over again or click the same buttons repeatedly. In addition to wasting time, sooner or later even the most careful researcher will lose focus while doing this and make mistakes.*

Solution: write (and save) scripts that execute repetitive tasks.

Makefiles are ideal for this.

Best practices

3. Make incremental changes

> *Unlike traditional commercial software developers, but very much like developers in open source projects or startups, scientific programmers usually don't get their requirements from customers, and their requirements are rarely frozen. In fact, scientists often can't know what their programs should do next until the current version has produced some results. This challenges design approaches that rely on specifying requirements in advance.*

Solution: use version control systems such as `Git` and push changes frequently.

Set small short term goals and submit frequent changes. Uses branches if necessary to make changes that are more experimental.

Best practices

4. **Don’t repeat yourself (or others)**

   Anything that is repeated in two or more places is more difficult to maintain. Every time a change or correction is made, multiple locations must be updated, which increases the chance of errors and inconsistencies. To avoid this, program- mers follow the DRY Principle, for "don’t repeat yourself", which applies to both data and code.

Solution: use header files that contain definitions that are used in multiple places.

Solution: if code is repeated more than once, put in a function. Then have different parts of the code call this function.

Best practices

5. Plan for mistakes

Mistakes are inevitable, so verifying and maintaining the validity of code over time is immensely challenging. The first line of defense is defensive programming. Experienced programmers add assertions to programs to check their operation, because experience has taught them that everyone (including their future self) makes mistakes. An assertion is simply a statement that something holds true at a particular point in a program; as the example below shows, assertions can be used to ensure that inputs are valid, outputs are consistent, and so on

```
def sqrt_newton(x):
    """
    This is Newton's method for
    solving f(s) = s^2 - x = 0
    """

    assert(x>=0)
```

Best practices

Don Knuth (1938–)

Premature optimization is the root of all evil

6. Optimize software only after it works correctly

> *Today's computers and software are so complex that even experts find it hard to predict which parts of any particular program will be performance bottlenecks. The most productive way to make code fast is therefore to make it work correctly, then determine whether it's actually worth speeding it up.*

Solution: after writing a correct code, use a **profiler** to determine the bottlenecks in your code. We will use **gprof** to do this.

Best practices

7. Document design and purpose, not mechanics

The following comment is not useful:

```
i = i + 1 # Increment the variable 'i' by one.
```

The following comment is more useful:

```
def sqrt_newton(x):
    """
    INPUT: x (a non-negative real number)
    OUPUT: x (a non-negative real number) that
       is an approximation to the square root of x

    This method uses Newton's method for
    solving f(s) = s^2 - x = 0.
    """
```

Best practices

7. Document design and purpose, not mechanics

The following comment is not useful:

```
i = i + 1 # Increment the variable 'i' by one.
```

The following comment is more useful:

```
def sqrt_newton(x):
    """
    INPUT: x (a non-negative real number)
    OUPUT: x (a non-negative real number) that
       is an approximation to the square root of x

    This method uses Newton's method for
    solving f(s) = s^2 - x = 0.
    """
```

8. Collaborate

In the same way that having manuscripts reviewed by other scientists can reduce errors and make research easier to understand, reviews of source code can eliminate bugs and improve readability.

Lab assignment: Stack

Develop a program with **Stack** (application to Bisection method), update Git, submit both source codes and screenshots.