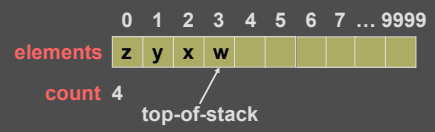
KIT107 PROGRAMMING

Data Structures and Algorithms
(Continued)

Dr Julian Dermoudy
& Dr Shuxiang Xu
School of ICT

1

5.3 Stack Implementation using Arrays



- **What would all of this look like in C (which is a *procedural* programming language)?**

2

4. Abstract Data Types

- **4.1 Abstract Data Types**
- **4.2 Representation**
- **4.3 Example: Modelling a Time**
- **4.4 ADTs in C**
- **4.5 Access Modifiers**

3

4.1 Abstract Data Types

- Program language independent concept
- Describe the structure of the data being manipulated
- Capture the relationships between different components of the data
- Encapsulates the operations available on the data with the data

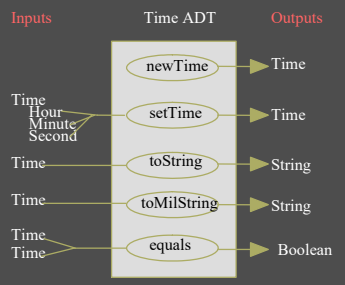
4

4.3 Example: Modelling a Time

- Pick a time of day
- What does it consist of?
  - an hour, a minute, and a second
- What can you do with a time value?
  - change it, share it (in AM/PM format and in 24hr format), and compare it with another time value
- How can you model the concept and implement it?

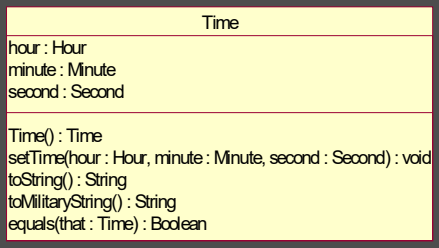
5

Example: Time (black box diagram)



6

Example: Time (UML diagram)



7

4.4 ADTs in C

- A C mechanism(s) is required that:
  - allows data and operations to be declared
  - enforces security
  - facilitates portability and code re-use
  - separates specification from implementation
  - encapsulates data and operations

8

Header File

- The C mechanism for the ADT specification is the header (.h) file
- Function *headings* (declarations) and types are included but nothing else

9

Header File

```
//time.h

#include <stdbool.h>

typedef struct {
        int hour;
        int minute;
        int second;
} time;

void setTime(time *tp, int h, int m, int s);
char *toString(time t);
char *toMilString(time t);
bool equals(time this, time that);
```


10

Aside: Where *didn't* the Time go?

- Q: The black box diagram showed the operations (`setTime()`, `toString()`, `toMilitaryString()`, and `equals()`) required a `Time` value as a parameter — it's in the header file too. Why?
- A: UML and Java are object-oriented — but C is *procedural*. You've got to give variables to functions, and not methods to objects.

11

C Implementation

- The C mechanism for the ADT implementation is the source (`.c`) file
- Global variables and function bodies (definitions) are included

12

Harness (Client) Files

- ADT implementations are 'passive' (library files)
- A client/harness program is required in order for the ADT to be used to achieve some task
- The same ADT can be used for many differing purposes, the client/harness file is different for each purpose

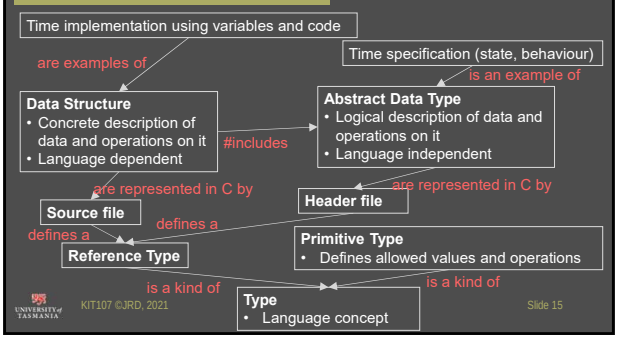
13

Example Harness

```
#include <stdio.h>
#include "time.h"

int main(int argc, char *argv[])
{
        time t;  /*declares t to be a Time variable*/

        setTime(&t,9,33,35);
        printf("The time is %s\n",toString(t));
}
```


14

Terminology Relationships



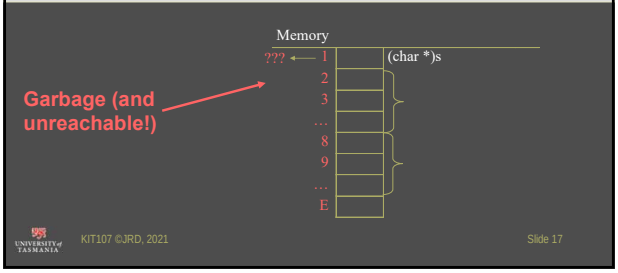
15

Garbage Collection

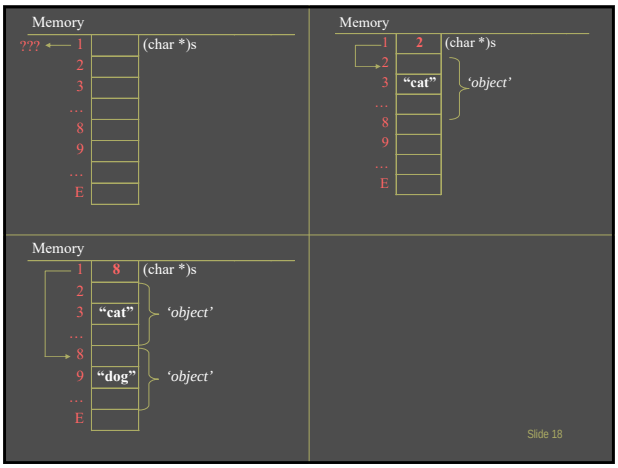
- Memory can become unreachable when pointer variables are assigned to a different address or deleted
- When this happens the value is *garbage* and a *space leak* has occurred
- C does not possess a *garbage collector* to free memory, memory must be `free()`d by the programmer

16

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
      char *s;
      s="cat";
      free(s);
      s="dog";
}
```



17



18

5. The Stack ADT

- 5.1 The Stack ADT
- 5.2 Genericity
- 5.3 Syntax vs Semantics
- 5.4 Primitive Operations vs Derived Operations
- 5.5 Stack Implementation in C (Using Arrays)

19

5.1 The Stack ADT

- A *Stack* is either empty (*null*) or consists of a first element (the *top-of-stack*) and the remainder of the stack which is itself a stack
- The Stack is a *recursive* (or *self-referential*) data structure

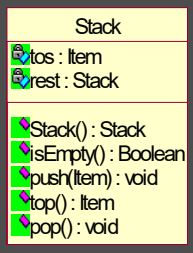
20

Stack Structure and Operations

- Last-In-First-Out structure
- Only *top of stack* visible
- Can only *push* new items onto top
- Can only *pop* items off the top
- Example: stack of plates, clothes on the floor, post-fix calculator

21

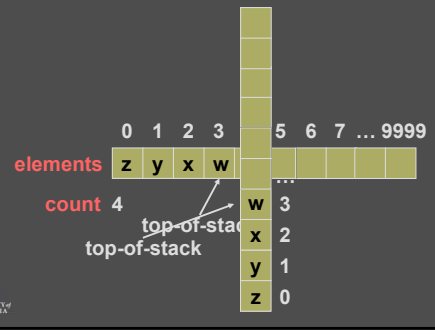
The Stack ADT Diagram



KIT107 ©JRD, 2021 Slide 22

22

5.6 Stack Implementation using Arrays



Slide 23

23

The Stack ADT C Header

```
#include <stdbool.h>

typedef struct {
        int count;
        char elements[10000];
} stack;

bool isEmpty(stack s);
void push(stack *sp,char i);
char top(stack s);
void pop(stack *sp);
```

KIT107 ©JRD, 2021 Slide 24

24

5.4 Polymorphism and Genericity

- This is a stack of `char`. How can any kind of item be represented?
- A type is needed which has a fixed size and a specified type…

25

The Stack ADT C Header

```
#include <stdbool.h>

typedef struct {
        int count;
        item elements[10000];
} stack;

bool isEmpty(stack s);
void push(stack *sp, item i);
item top(stack s);
void pop(stack *sp);
```


26

Polymorphism and Genericity (continued)

- The solution is to use `(void *)`. This is of fixed type and can be cast/coerced to any type required
- Thus a stack of `(void *)`s is *generic*
- (This brings two problems: consistency of type content and necessity for type casting)

27

The Stack ADT C Header

```
#include <stdbool.h>

typedef struct {
        int count;
        void *elements[10000];
} stack;

bool isEmpty(stack s);
void push(stack *sp,void *i);
void *top(stack s);
void pop(stack *sp);
```


28

5.5 Opacity and Initialisation

- The current version has two other problems. The 'client' programmer has
  - Access to the internal representation of the ADT from the included header file
  - To create the stack knowing the internals and then guess at how to initialise it

29

Opacity and Initialisation (continued)

- The solution is to use a forward declaration and a pointer, move the internals to the source file, and define a function which builds and initialises the ADT

30

The Stack ADT C Header

```
struct stack_int;
typedef struct stack_int *stack;

void init_stack(stack *sp);
bool isEmpty(stack s);
void push(stack s,void *i);
void *top(stack s);
void pop(stack s);
```


31

5.6 Primitive Operations vs Derived Operations

- All the preceding operations are *primitive operations* — they cannot be implemented without knowledge of the underlying implementation
- Those operations not requiring information of the underlying implementation are called *derived operations*

32

Example

- Derived operations:

```
height(Stack s)  = 0, if isEmpty(s)
                 = 1 + height(pop(s)), otherwise

join(Stack s1, Stack s2)   = s2, if isEmpty(s1)
                           = push(top(s1),join(pop(s1),s2)),
                             otherwise
```


33

6. Linked Lists

- 6.1 Linked List Concepts
- 6.2 Stack Implementation
- 6.3 Example Application
- 6.4 Arrays vs Linked Lists

34

Aeroplanes!

- Planes: fixed size, size is specified at construction time, plane contains seats only

35

Trains!

- Trains: variable size, size can grow/ shrink as necessary, train contains seats plus couplings

36

6.1 Linked List Concepts

- Planes versus Trains
- Arrays versus Linked Lists
  - (also direct versus sequential access)

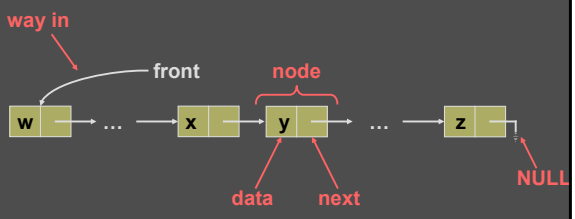
37

Linked Lists (continued)

- A *linked list* is a sequence of self-referential components called *nodes*
- Each node contains a data field (traditionally called data) and a uni-directional coupling field (traditionally called *next*)
- The end of the list is indicated by the sentinel address `NULL`

38

Example



39

6.2 Stack Implementation using Linked Lists

- The specification (black box, UML diagram, and C header) don't change
- There is no need for the array global variable or its length variable (and if you didn't care about consistency/transparency, you could also eliminate the `struct`!)
- There is a need for the node type and associated global variable

40

Stack Implementation using Linked Lists

- The implementation will use (`void *`) to represent 'anything'
- The literal value `NULL` indicates that a reference variable refers to no value, i.e. it is the end of the linked list

41

6.3 Example Application

- Reversing a string (or indeed a train!) can easily be achieved through the use of a stack

42

6.4 Arrays versus Linked Lists

- Array
  - fixed, specified size
  - implicit ordering by index
  - no space overhead for ordering
  - direct access to all elements
  - insertion and deletion require shuffling
- Linked List
  - variable size
  - explicit ordering using references
  - space overhead for ordering (next field)
  - sequential access from first element
  - insertion and deletion in place

43

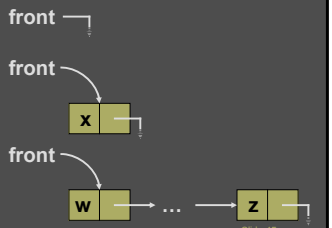
7. Linked List Operations

- 7.1 Emptiness Testing
- 7.2 Traversal
- 7.3 Modification
- 7.4 Insertion
- 7.5 Deletion
- 7.6 Polymorphism

44

7. Linked List Operations

- Three forms of list:
  - empty list (A)
  - singleton list (B)
  - general list (C)



45

Linked List Operations (continued)

- Operations:
  - emptiness testing
  - traversal
  - modification of an element
  - insertion
    - at front
    - at rear (append)
    - between
  - deletion
    - from front
    - from rear
    - between

46

Linked List Operations (continued)

- For some linked list type `list` declared as before…

```
struct list_int;
typedef struct list_int *list;

struct list_int {
  node front;
};
```


47

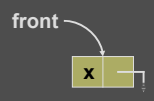
7.1 Emptiness Testing — A

```
bool isEmpty(list l)
{
  return (l->front == NULL);
}
```

48

7.1 Emptiness Testing — B

```
bool isEmpty(list l)
{
   return (l->front == NULL);
}
```



49

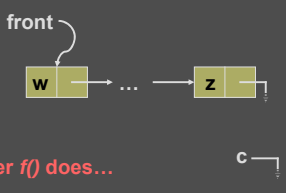
7.1 Emptiness Testing — C


```
bool isEmpty(list l)
{
   return (l->front == NULL);
}
```

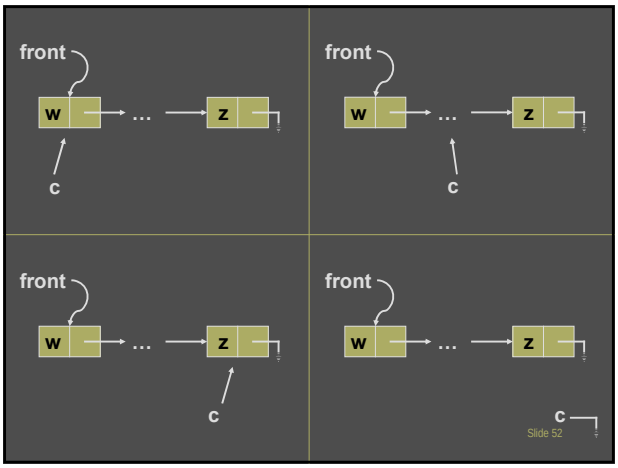

50

7.2 Traversal — C

```
void traverse(list l)
{
   node c;

   c=l->front;
   while (c != NULL) {
       f(getData(c));
       c=getNext(c);
   }
}
```



51



52

7.2 Traversal — A

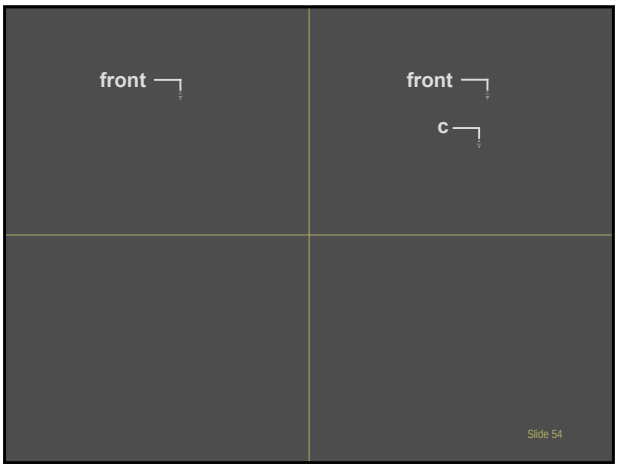
```
void traverse(list l)
{
   node c;

   c=l->front;
   while (c != NULL) {
      f(getData(c));
      c=getNext(c);
   }
}
```

front

c

Whatever *f()* does…

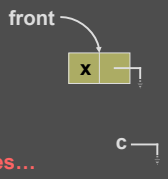
53



54

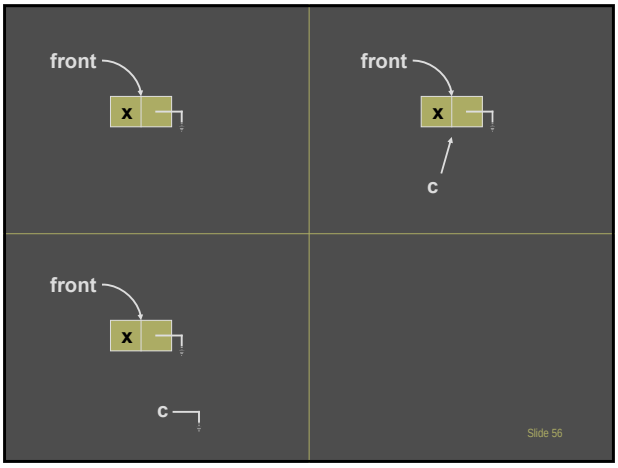
7.2 Traversal — B

```
void traverse(list l)
{
   node c;

   c=l->front;
   while (c != NULL) {
      f(getData(c));
      c=getNext(c);
   }
}
```



Whatever *f()* does…

55



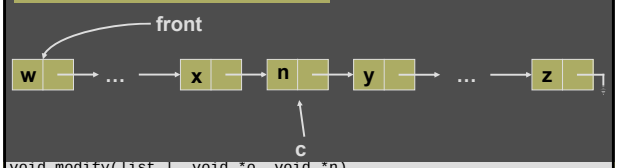
56

7.3 Modification — A

```
void modify(list l, void *o, void *n)
{
   node c;

   if (isEmpty(l))
   {
      fprintf(stderr,"list is empty.");
      exit(1);
   }
   else
   {
      …
}
```

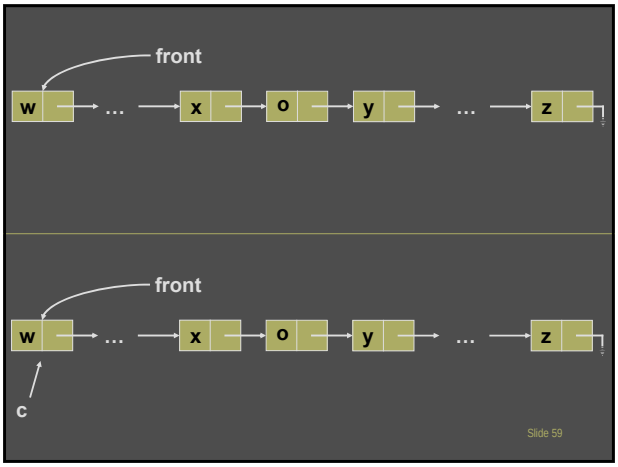
57

7.3 Modification — C

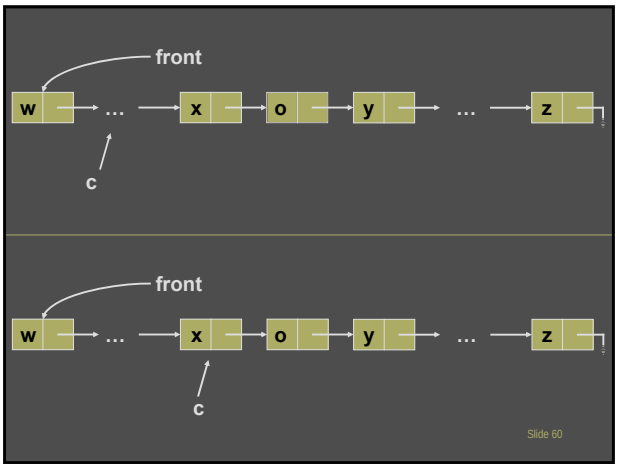



```
void modify(list l, void *o, void *n)
{
   …
      c = l->front;
      while ((c!=NULL) && (! equals(o,getData(c))) c=getNext(c);
      if (c == NULL) exit(1); else setData(c, n);
   }
}
```

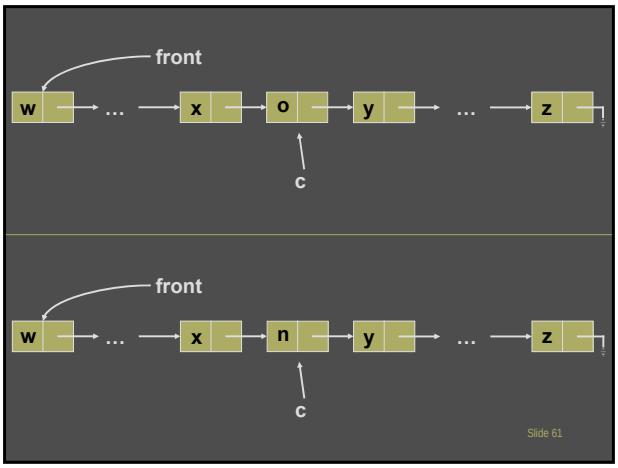
58



59



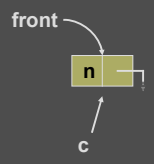
60



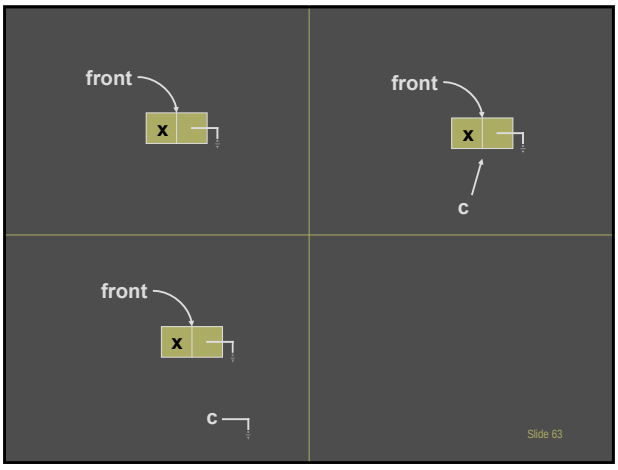
61

7.3 Modification — B

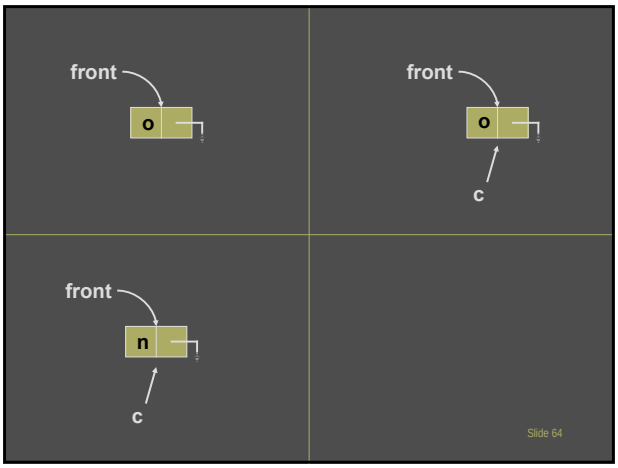
```
void modify(list l, void *o, void *n)
{
   …
      c = l->front;
      while ((c != NULL) &&
             (! equals(o, getData(c)))
         c = getNext(c);
      if (c == NULL)
      {
         fprintf(stderr, "list is empty.");
         exit(1);
      }
      else
         setData(c, n);
   }
}
```



62



63



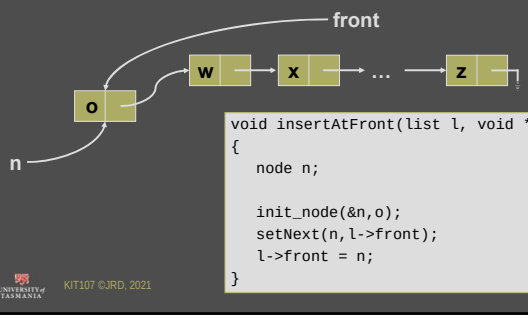
64

7.4 Insertion

- 7.4.1 Insertion at Front
- 7.4.2 Insertion at Rear
- 7.4.3 Insertion in Middle

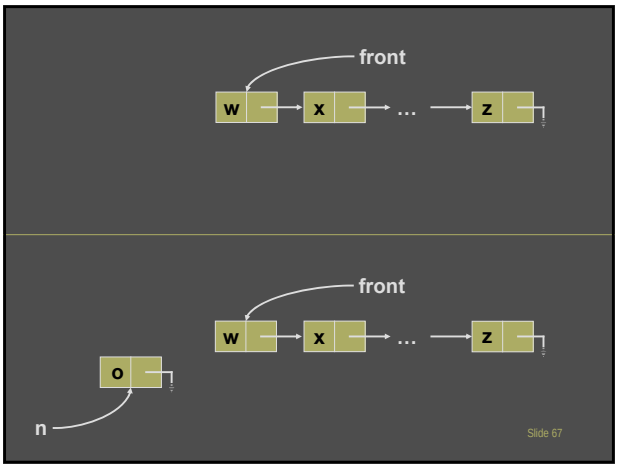
65

7.4.1 Insertion at Front — C

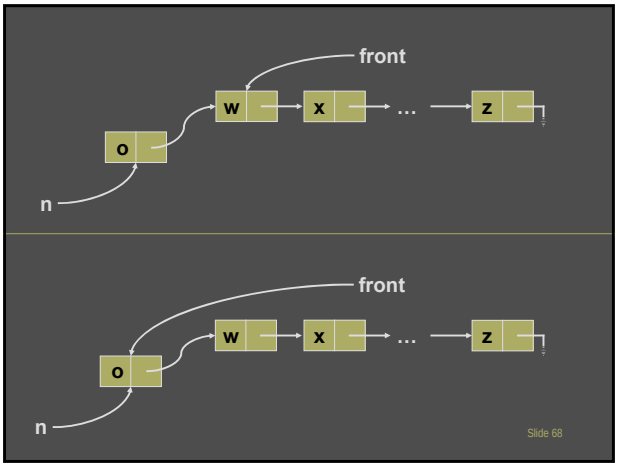



```
void insertAtFront(list l, void *o)
{
   node n;

   init_node(&n,o);
   setNext(n,l->front);
   l->front = n;
}
```


66



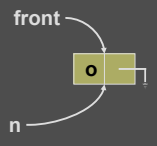
67



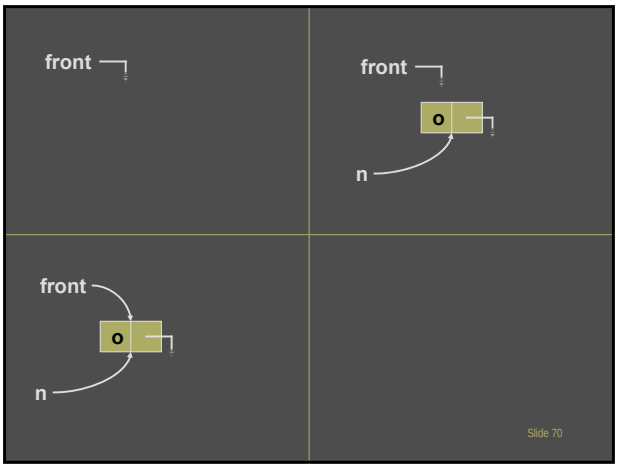
68

7.4.1 Insertion at Front — A

```
void insertAtFront(list l, void *o)
{
   node n;

   init_node(&n,o);
   setNext(n,l->front);
   l->front = n;
}
```



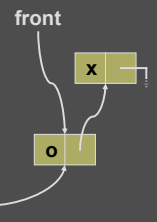
69



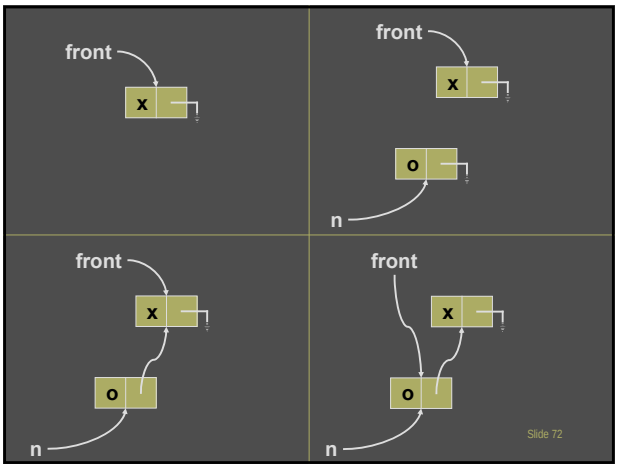
70

7.4.1 Insertion at Front — B

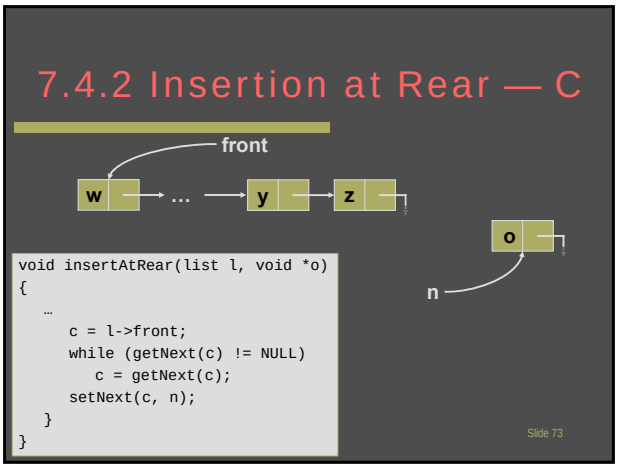
```
void insertAtFront(list l, void *o)
{
   node n;

   init_node(&n,o);
   setNext(n,l->front);
   l->front = n;
}
```



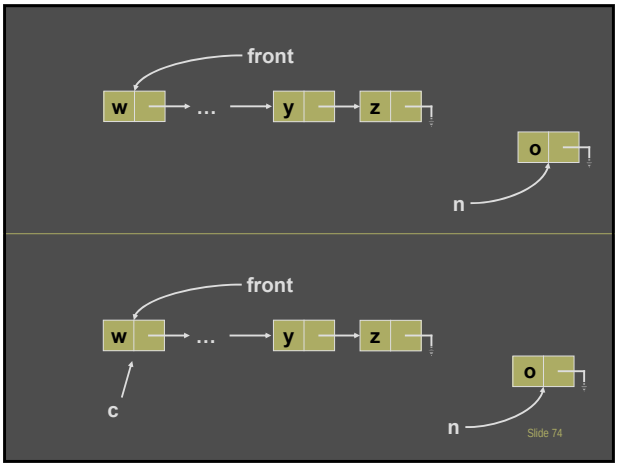
71



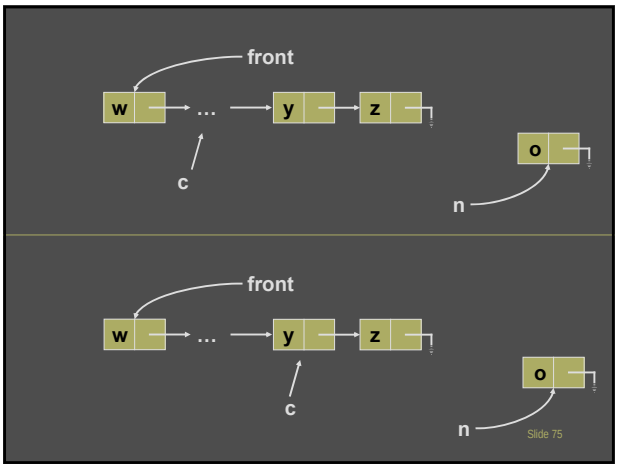
72


7.4.2 Insertion at Rear — C
front
w
…
y
z
o
n
void insertAtRear(list l, void *o)
{
…
c = l->front;
while (getNext(c) != NULL)
c = getNext(c);
setNext(c, n);
}
}
Slide 73

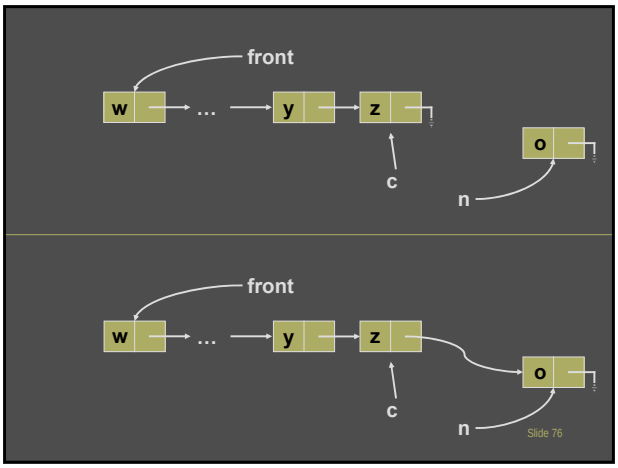
73


front
w
…
y
z
o
n
front
w
…
y
z
c
o
n
Slide 74

74


front
w
…
y
z
c
o
n
front
w
…
y
z
c
o
n
Slide 75

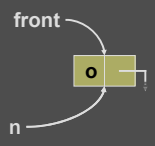
75



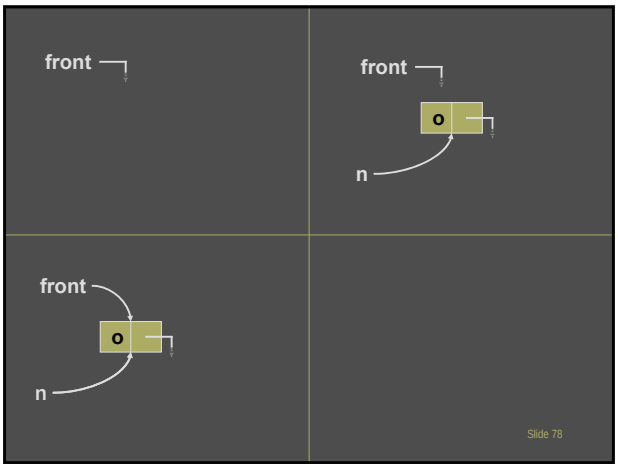
76

7.4.2 Insertion at Rear — A

```
void insertAtRear(list l, void *o)
{
   node n, c;

   init_node(&n, o);
   if (isEmpty(l))
      l->front = n;
   else
   {
      c = l->front;
      while (getNext(c) != NULL)
         c = getNext(c);
      setNext(c, n);
   }
}
```



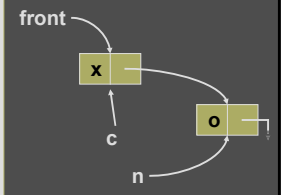
77



78

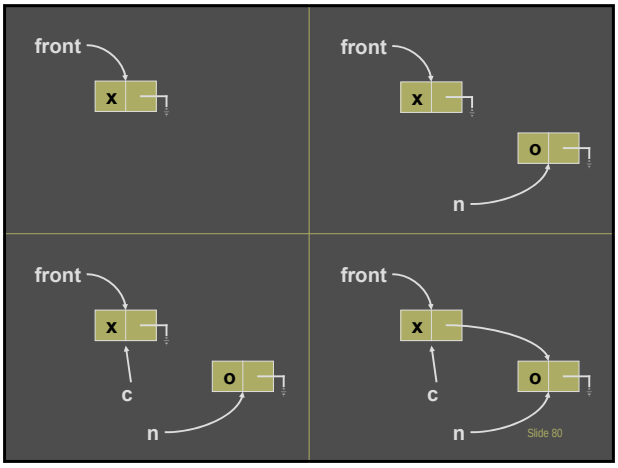
7.4.2 Insertion at Rear — B

```
void insertAtRear(list l, void *o)
{
   node n, c;

   init_node(&n, o);
   if (isEmpty(l))
      l->front = n;
   else
   {
      c = l->front;
      while (getNext(c) != NULL)
         c = getNext(c);
      setNext(c, n);
   }
}
```



Slide 79

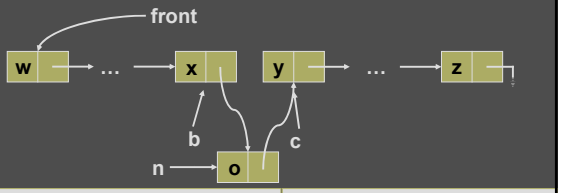
79



Slide 80

80

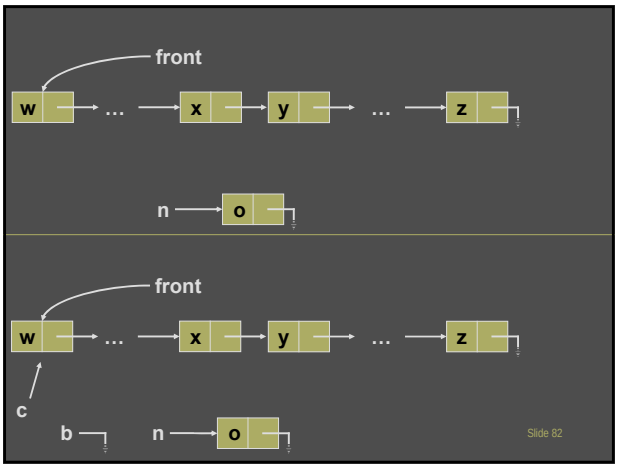
7.4.3 Insertion in Middle — C



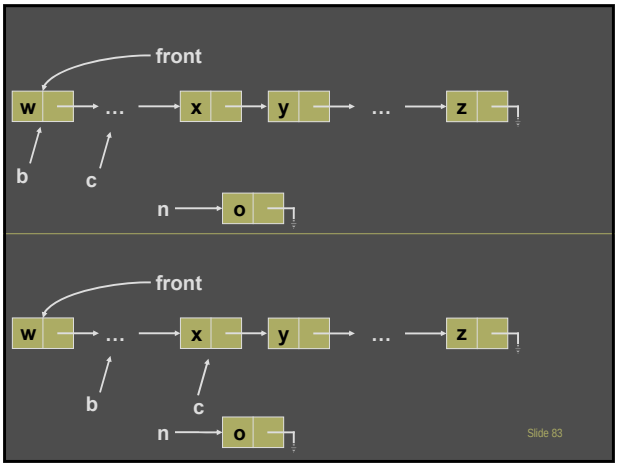

```
void insertBetween(list l, void *o)
{
      …
      b = NULL;
      c = l->front;
      while (!done)
      {
         b = c;
         c = getNext(c);
      }
      if (b == NULL)
         l->front = n;
      else setNext(b, n);
      setNext(n, c);
   }
}
```

Whatever *done* is…

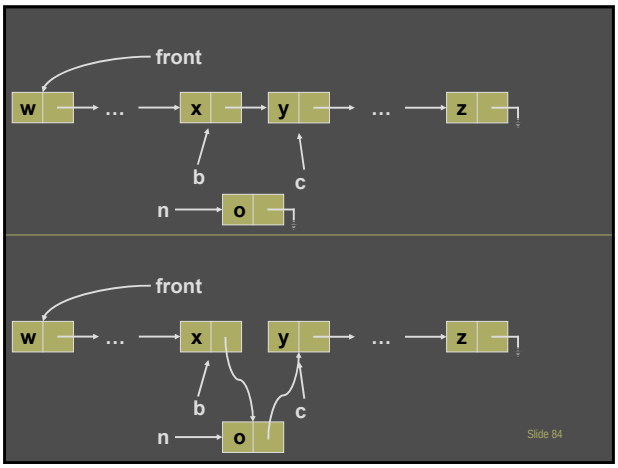
81


front
w
…
x
y
…
z
n
o
front
w
…
x
y
…
z
c
b
n
o
Slide 82

82


front
w
…
x
y
…
z
b
c
n
o
front
w
…
x
y
…
z
b
c
n
o
Slide 83

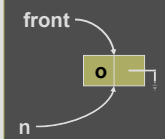
83


front
w
…
x
y
…
z
b
c
n
o
front
w
…
x
y
…
z
b
c
n
o
Slide 84

84

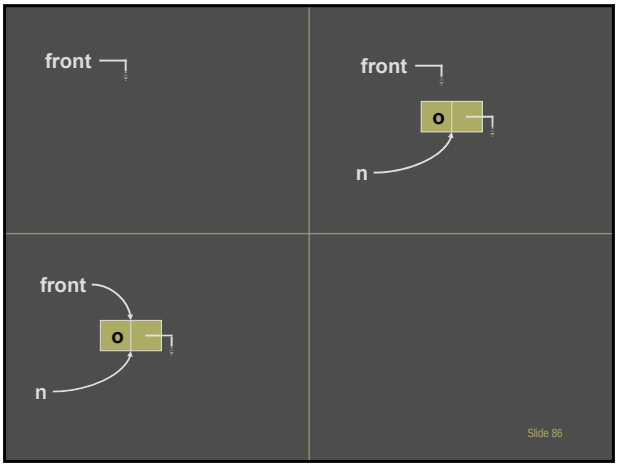
7.4.3 Insertion in Middle — A

```
void insertBetween(list l, void *o)
{
   node n, c, b;

   init_node(&n, o);
   if (isEmpty(l))
      l->front = n;
   else
   {
      …
   }
}
```



Slide 85

85

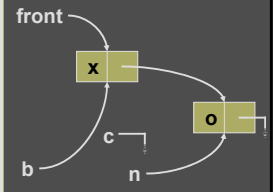


86

7.4.3 Insertion in Middle — B

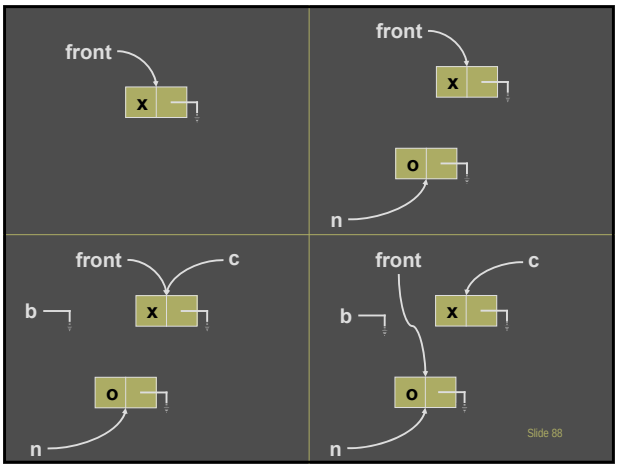
```
void insertBetween(list l, void *o)
{
      …
      b = NULL;
      c = l->front;
      while (!done)
      {
         b = c;
         c = getNext(c);
      }
      if (b == NULL) l->front = n;
      else setNext(b, n);
      setNext(n, c);
   }
}
```

Whatever *done* is…

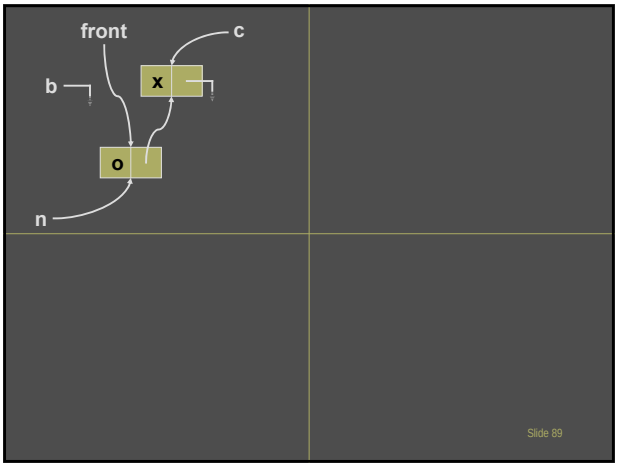


Slide 87

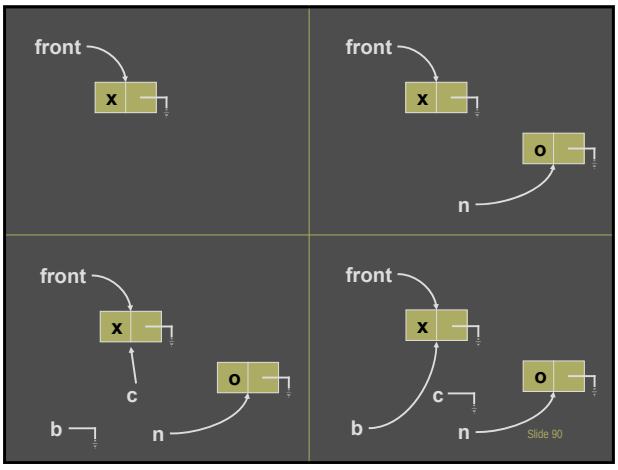
87


front
x
front
x
o
n
front
c
b
x
o
n
front
c
b
x
o
n
Slide 88

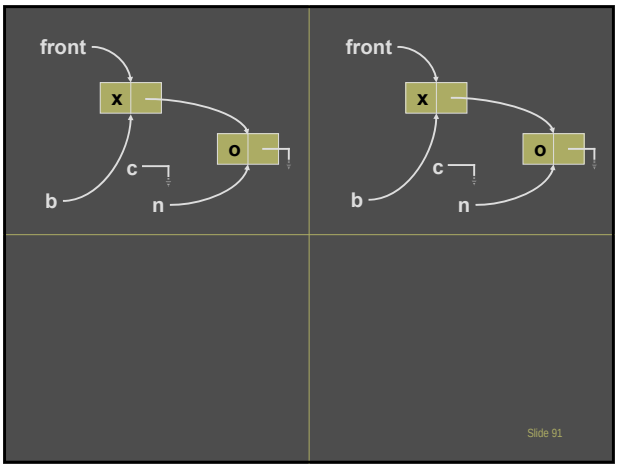
88


front
c
b
x
o
n
Slide 89

89


front
x
front
x
o
n
front
x
c
b
o
n
front
x
c
b
o
n
Slide 90

90



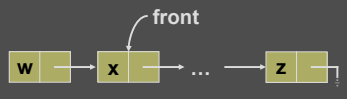
91

7.5 Deletion

- 7.5.1 Deletion from Front
- 7.5.2 Deletion from Back
- 7.5.3 Deletion from Middle

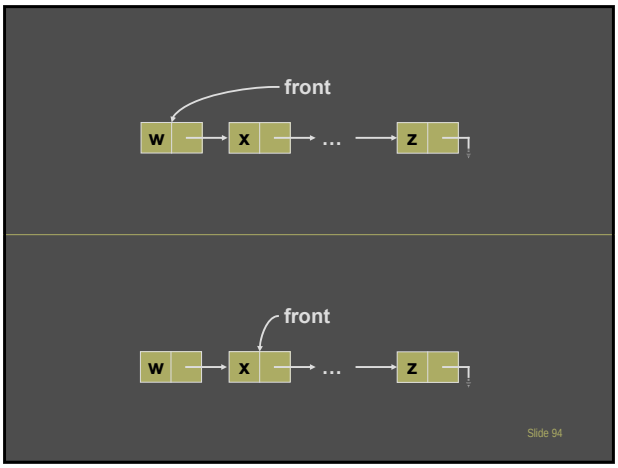
92

7.5.1 Deletion from Front — C




```
void removeFromFront(list l)
{
   if (isEmpty(l))
      …
   else
      l->front = getNext(l->front);
}
```


93



94

7.5.1 Deletion from Front — A

```
void removeFromFront(list l)
{
   if (isEmpty(l))
   {
      fprintf(stderr, "list is empty.");
      exit(1);
   }
   else
      l->front = getNext(l->front);
}
```

front

95

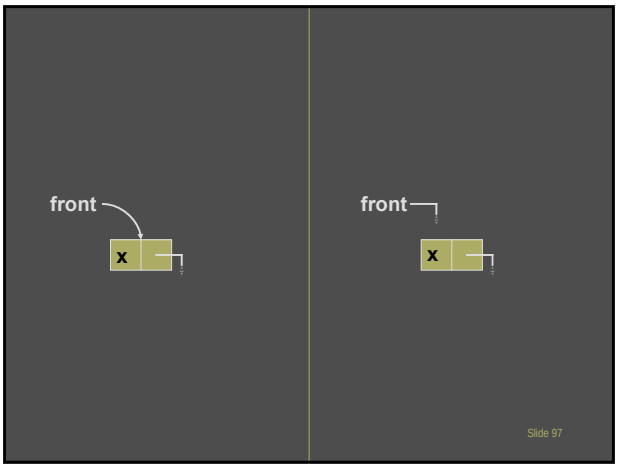
7.5.1 Deletion from Front — B

```
void removeFromFront(list l)
{
   if (isEmpty(l))
   {
      fprintf(stderr, "list is empty.");
      exit(1);
   }
   else
      l->front = getNext(l->front);
}
```

front

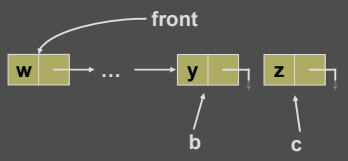
x

96



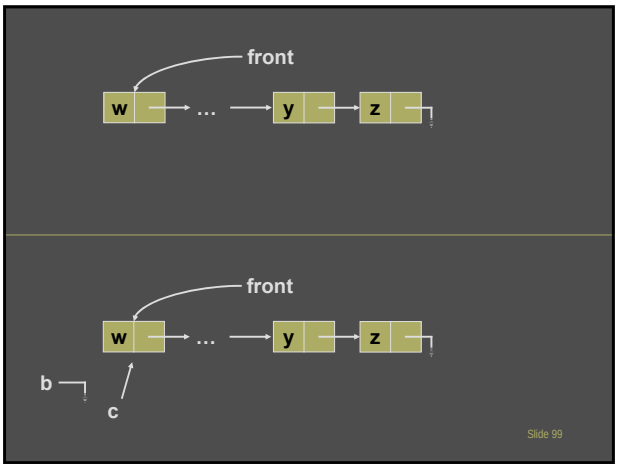
97

7.5.2 Deletion from Back — C

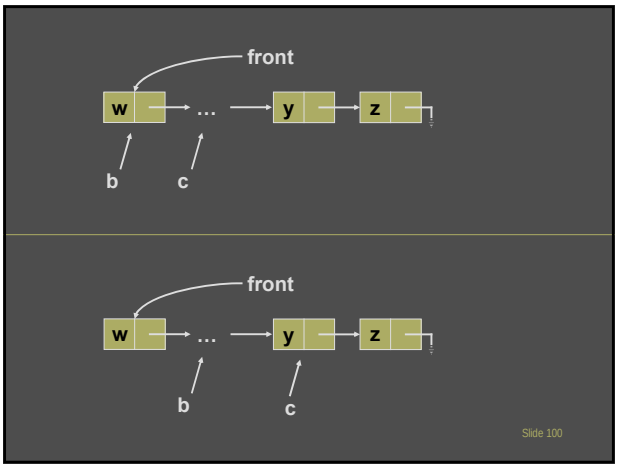



```
void removeFromBack(list l)
{
      …
      b = NULL; c = l->front;
      while (getNext(c) != NULL)
      {
         b = c; c = getNext(c);
      }
      if (b == NULL)
         l->front = NULL;
      else
         setNext(b, NULL);
   }
}
```

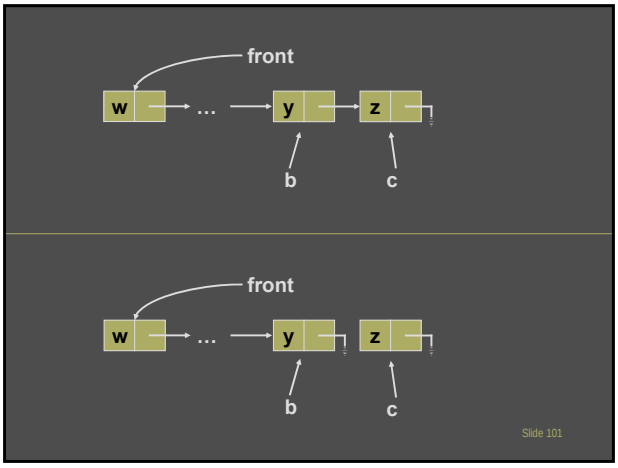
98



99



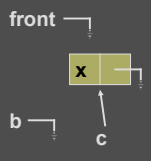
100



101

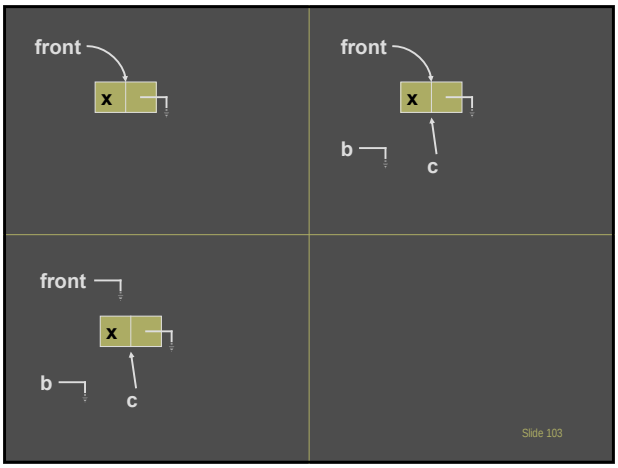
7.5.2 Deletion from Back — B

```
void removeFromBack(list l)
{
      …
      b = NULL;
      c = l->front;
      while (getNext(c) != NULL)
      {
         b = c;
         c = getNext(c);
      }
      if (b == NULL)
         l->front = NULL;
      else
         setNext(b, NULL);
   }
}
```



Slide 102

102



103

7.5.2 Deletion from Back — A

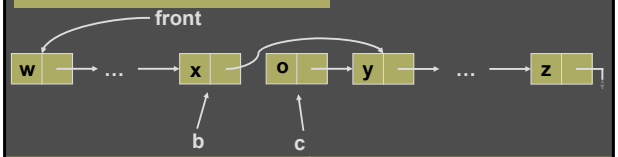
```
void removeFromBack(list l)
{
   node b, c;

   if (isEmpty(l))
   {
      fprintf(stderr, "list is empty.");
      exit(1);
   }
   else
   {
      …
   }
}
```

front

Slide 104

104

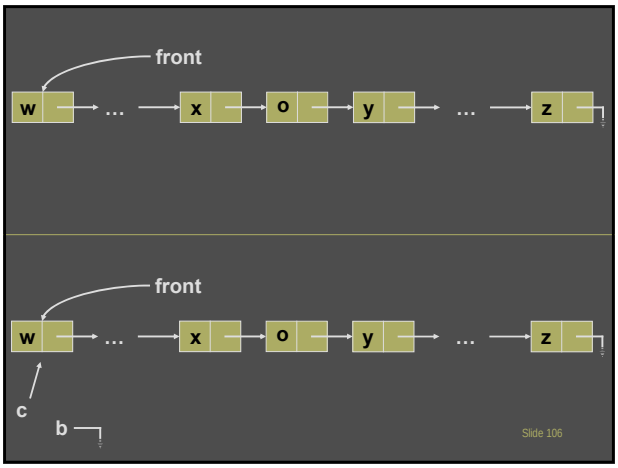
7.5.3 Deletion from Middle — C



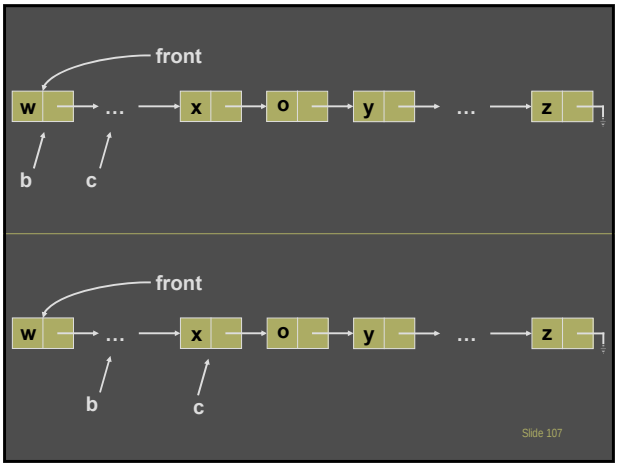

```
void removeFromMiddle(list l)
{
   …
      b = NULL;
      c = l->front;
      while (!done)
      {
         b = c;
         c = getNext(c);
      }
      if (b == NULL)
         l->front = getNext(c);
      else
         setNext(b, getNext(c));
   }
}
```

Whatever *done* is …

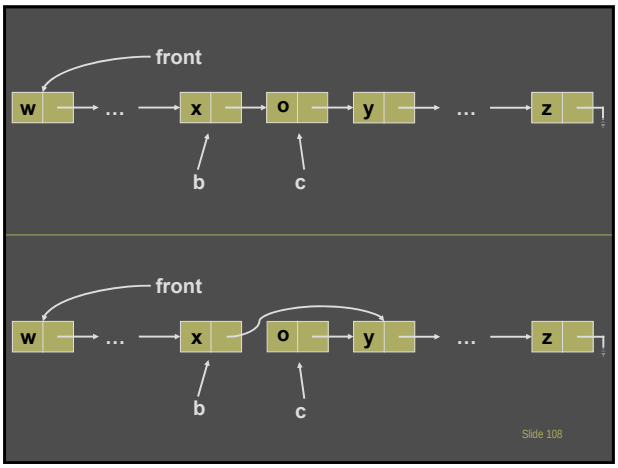
105


front
w
…
x
o
y
…
z
front
w
…
x
o
y
…
z
c
b
Slide 106

106


front
w
…
x
o
y
…
z
b
c
front
w
…
x
o
y
…
z
b
c
Slide 107

107

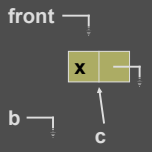

front
w
…
x
o
y
…
z
b
c
front
w
…
x
o
y
…
z
b
c
Slide 108

108

7.5.3 Deletion from Middle — B

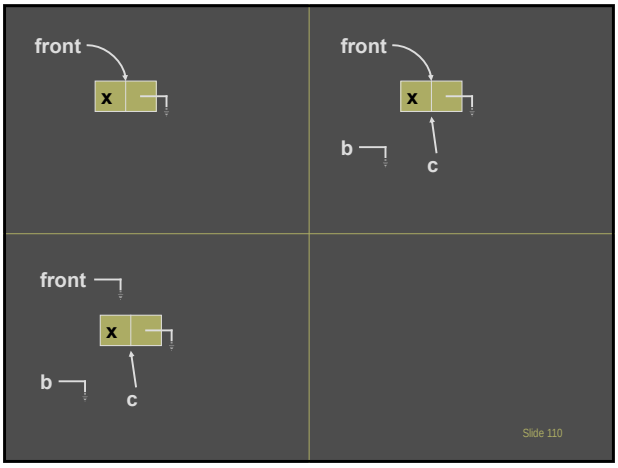
```
void removeFromMiddle(list l)
{
   …
      b = NULL;
      c = l->front;
      while (!done)
      {
         b = c;
         c = getNext(c);
      }
      if (b == NULL)
         l->front = getNext(c);
      else
         setNext(b, getNext(c));
   }
}
```

Whatever *done* is...



Slide 109

109



110

7.5.3 Deletion from Middle — A

```
void removeFromMiddle(list l)
{
   node b, c;

   if (isEmpty(l))
   {
      fprintf(stderr, "list is empty.");
      exit(1);
   }
   else
   {
      …
   }
}
```

Slide 111

111

7.6 Polymorphism

- Recall traverse…
- Q: How can f( ) be specified so that it is whatever we need it to be?

```
void traverse(list l)
{
   node c;

   c=l->front;
   while (c != NULL) {
      f(getData(c));
      c=getNext(c);
   }
}
```

Whatever *f()* does…

KIT107 ©JRD, 2021 Slide 112

112

Polymorphism (continued)

- Q: How can f( ) be specified so that it is whatever we need it to be?
- A: Pointers!

```
void traverse(list l,
   void (*fp)(void *))
{
   node c;

   c=l->front;
   while (c != NULL) {
      (*fp)(getData(c));
      c=getNext(c);
   }
}
```

KIT107 ©JRD, 2021 Slide 113

113

Polymorphism (continued)

- To use, we just write the function we want and pass its address as a parameter…

```
void print(void *o)
{
   printf("%s\n", (char *)o);
}

…
traverse(l, (&print));
…
```

KIT107 ©JRD, 2021 Slide 114

114

Polymorphism (continued)

- We can do the same thing for `equals()` in `modify()`...

```
void modify(list l, void *o, void *n,
      bool (*equals)(void *,void *))
{
      …
      while ((c != NULL) &&
                  (! ((*equals)(o, getData(c)))))
      …
}
```

115

Polymorphism (continued)

- And the definition and use might be:

```
bool same(void *a, void *b)
{
   return (strcmp(a, b) == 0);
}

…
modify(l, "one", "four", &same);
…
```


116

8. The Queue ADT

- 8.1 The Queue ADT
- 8.2 Queue Implementation in C (Using Linked Lists)
- 8.3 Queue Implementation in C (Using Arrays)
- 8.4 Queue Example

117

8.1 The Queue ADT

- A *Queue* is either empty (*null*) or consists of a first element (the *head/front*) and the remainder of the queue (the *tail*) which is itself a queue
- The Queue is a *recursive* (or *self-referential*) data structure

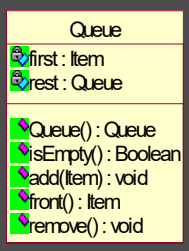
118

The Queue ADT (continued)

- First-In-First-Out structure
- Only *front of queue* visible
- Can only *add* new items onto rear
- Can only *remove* items from the front
- Examples: traffic intersection, bus stop, bank, cashier

119

The Queue UML Diagram



120

The Queue ADT C Header

```
struct queue_int;
typedef struct queue_int *queue;

void init_queue(queue *qp);
bool isEmpty(queue q);
void add(queue q, void *i);
void *front(queue q);
void rear(queue q);
```

121

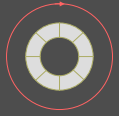
8.2 Queue Implementation in C (Using Linked Lists)

- **Exercise: see tutorial**

122

8.3 Queue Implementation in C (Using Arrays)

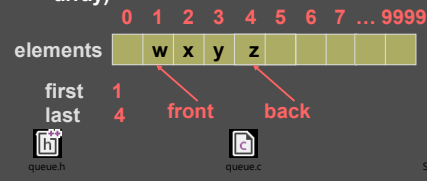
- **Two alternative solutions are presented here:**



123

8.3.1 Array Implementation (version 1)

- Bounded implementation :
  - advance front-of-queue index (which will repeatedly shrink the size of the available array)

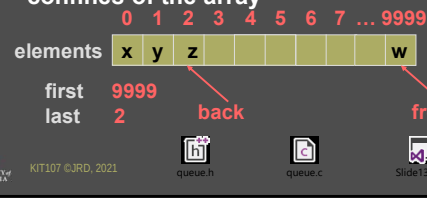


queue.h queue.c Slide134.sln

124

8.3.2 Array Implementation (version 2)

- Wrapped implementation allow the front- and back-of-queue indices to wrap around the ends of the array thus providing a 'virtual queue' within the physical confines of the array



queue.h queue.c Slide135.sln

125

8.4 Queue Example

- Driving across a road intersection can be achieved through the use of a queue

harness.c Slide134.sln Slide135.sln

126

9. Searching

- 9.1 Linked List Searching
- 9.2 Array Searching

127

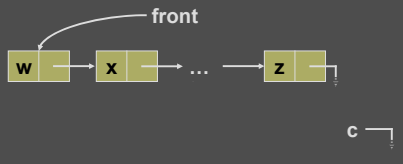
Searching

- Linear search if only sequential access is possible
- Binary search if direct access is possible and the data are sorted

128

9.1 Linked List Searching

- Can only access the first element and then move sequentially through the elements in the list



129

Time Taken for Linear Search

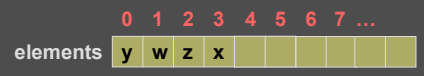
- The best case — searching for the first node
  $B(n) = 1$
- The worst case — searching for the last node
  (n is the size of the list)
  $W(n) = n$
- Average case

$$A(n) = \frac{1 + 2 \ldots\ldots + n}{n} = \frac{n + 1}{2}$$

130

9.2 Array Searching

- Unsorted array:
  - Step through the array to find a particular item
  - The time spent is the same as the case of linked list



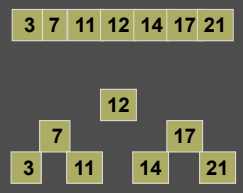
131

Array Searching

- Sorted array:
  - Start in the middle of the used portion
  - (*) Get the value of the 'current' element
  - Compare it to the required value
    - if equal stop
    - if current value is smaller discard bottom half of remaining array, go back to (*)
    - discard top half of remaining arrays, go back to (*)

132

Array Searching (continued)



- There are $\log_2(n+1)$ layers

133

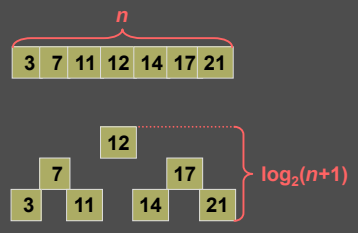
Aside: Logarithms!

- 4 + 5 = 9
- The inverse of + is –
- 9 – 4 = 5
- The symbols change and the order of operands changes

- $2 \times 2 \times 2 = 2^3 = 8$
- The inverse of exponentiation is the logarithm
- $\log_2 8 = 3$
- … just another …metic operator

134

Array Searching (continued)



- There are $\log_2(n+1)$ layers (approximately)

135

Time Taken for Binary Search

- The best case — searching for the middle element
  $B(n) = 1$
- The worst case — searching for an element at the lowest level
  $W(n) = \log_2(n+1)$
- The average case — searching for an element at mid depth
  $A(n) = \log_2(n+1)-1$

136

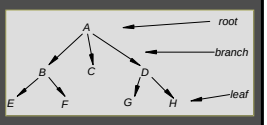
10. Trees

- 10.1 Introduction
- 10.2 Binary Trees
- 10.3 Tree Traversal
- 10.4 Binary Search Trees
- 10.5 General Search Trees

137

10.1 Introduction

- Trees are hierarchical structures
- Relationships exist:
  - *parent-child*
  - *siblings*
  - *ancestor-descendant*



- The ancestor of all is the *root*
- A node without a child is a *leaf*
- The number of child nodes is the *degree*

138

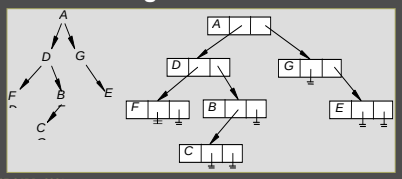
10.2 Binary Trees

- A tree of degree 2 is a *binary tree*
- Branches are called *left* and *right*
- A binary tree has the form:
  - empty, or
  - a node with a value and two branches (each of which is a binary tree)

139

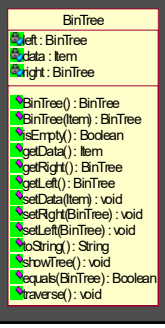
Structure

- Conceptually, a binary tree may be drawn as below on the left
- When implemented in C it appears as below on the right



140

The Binary Tree UML Diagram



141

BTNode and BinTree Classes

```
struct btnode_int;
typedef struct btnode_int *btnode;

struct btnode_int {
    void *data;
    btnode left;
    btnode right;
};
```

KIT107 ©JRD, 2021 BTNode.h BTNode.c BinTree.h BinTree.c Lab152.sln Slide 142

142

Alternative Node Definitions

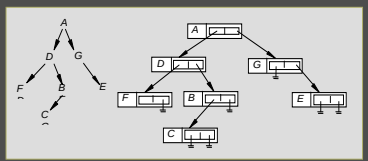
- One variable per branch (`left` and `right`) is fine for a small number of branches
- Two alternatives:
  - array of branches
  - linked list of branches

KIT107 ©JRD, 2021 Slide 143

143

Binary Tree using An Array

- First alternative
  - array of branches



KIT107 ©JRD, 2021 Slide 144

144

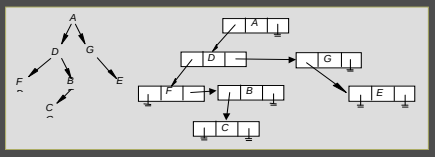
Binary Tree using An Array

```
struct btnode_int;
typedef struct btnode_int *btnode;

struct btnode_int {
    void *data;
    btnode branches[];
};
```


145

Binary Tree using References

- **Second alternative**
  - **linked list of branches**



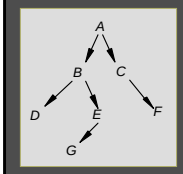
146

Binary Tree using References

```
struct btnode_int;
typedef struct btnode_int *btnode;

struct btnode_int {
    void *data;
    btnode child;
    btnode sibling;
};
```


147

10.3 Tree Traversal

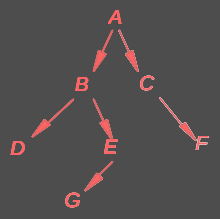


- Node first, then branches: ABDEGCF (*depth-first pre-order*)
- Left first, then node, then right: DBGEACF (*depth-first in-order*)
- Branches first, then node: DGEBFCA (*depth-first post-order*)
- In levels: ABCDEFG (*breadth-first*)

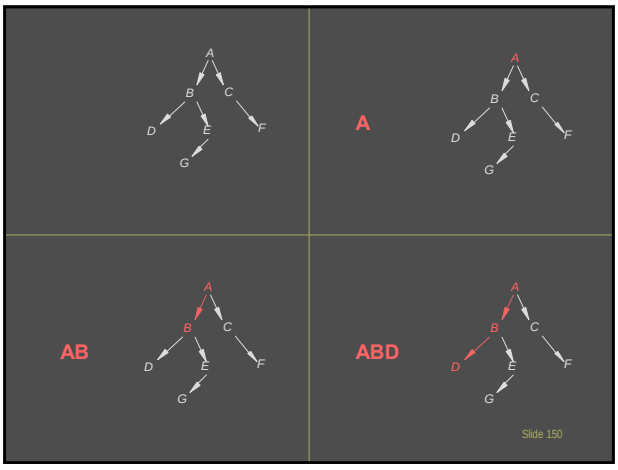
148

10.3.1 Depth-First Pre-Order Traversal

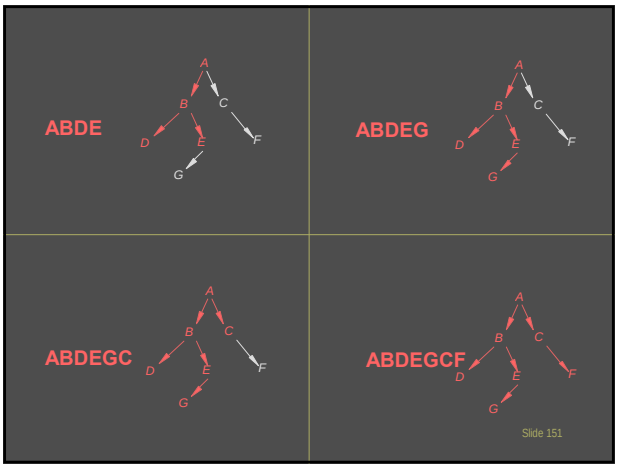
ABDEGCF



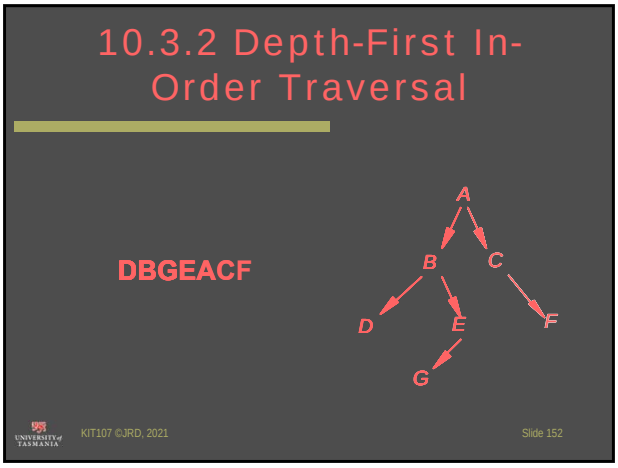
149



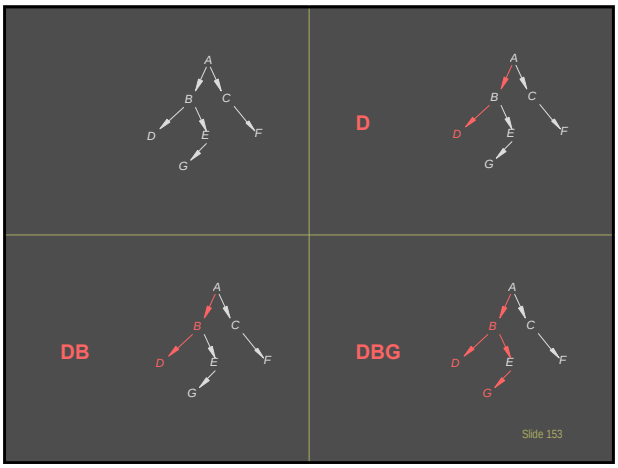
150


ABDE
ABDEG
ABDEGC
ABDEGCF
Slide 151

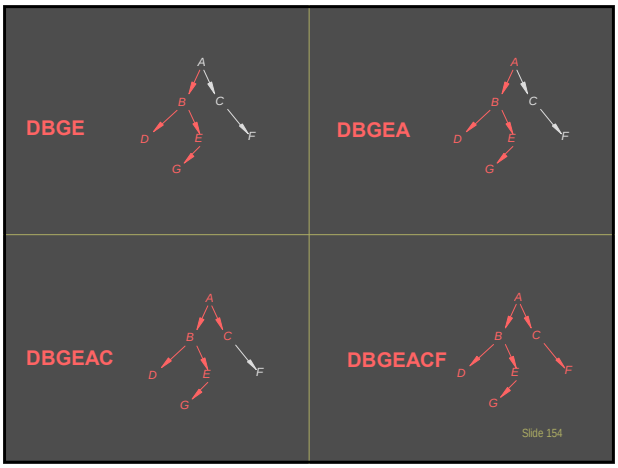
151


10.3.2 Depth-First In-Order Traversal
DBGEACF
KIT107 ©JRD, 2021
Slide 152

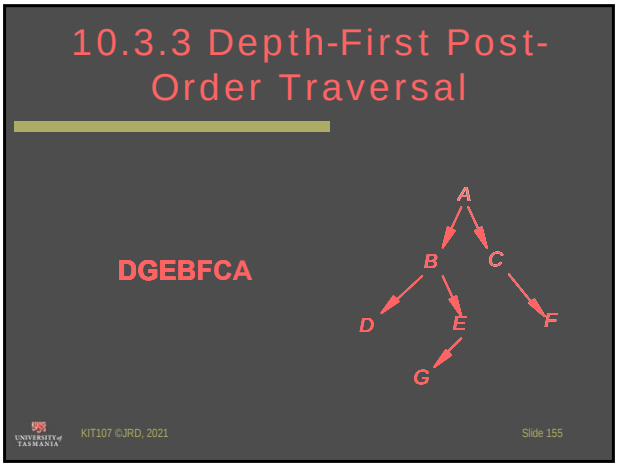
152


D
DB
DBG
Slide 153

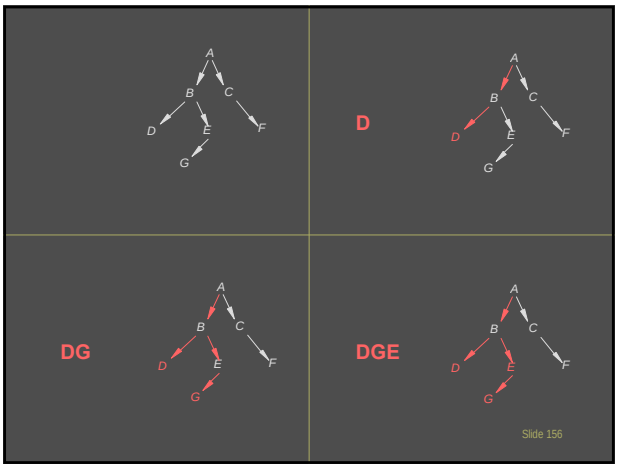
153


DBGE
DBGEA
DBGEAC
DBGEACF
Slide 154

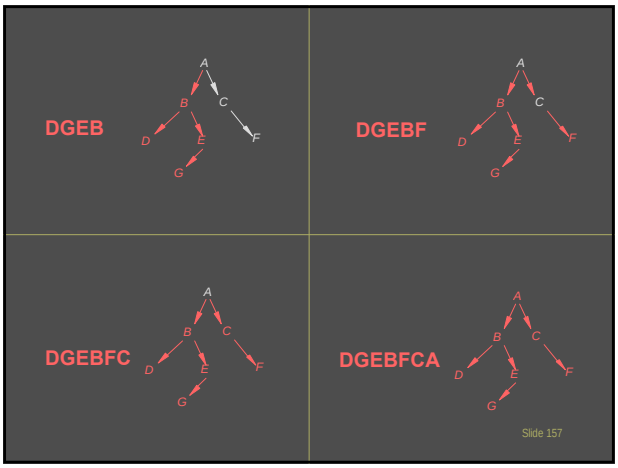
154


10.3.3 Depth-First Post-Order Traversal
DGEBFCA
KIT107 ©JRD, 2021
Slide 155

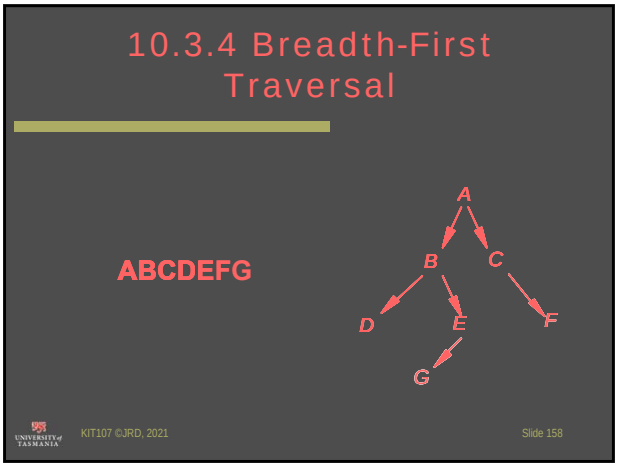
155


D
DG
DGE
Slide 156

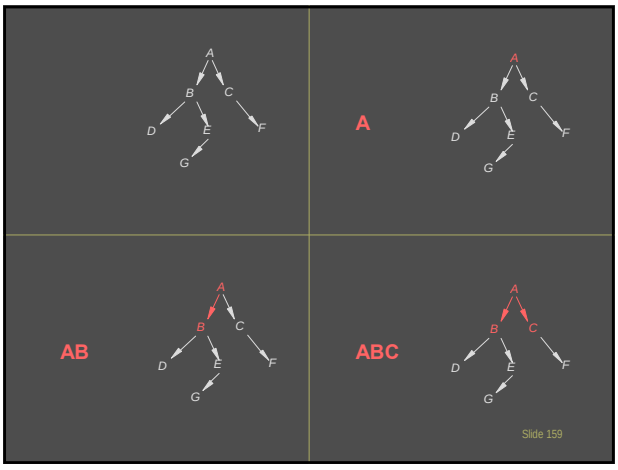
156


DGEB
DGEBF
DGEBFC
DGEBFCA
Slide 157

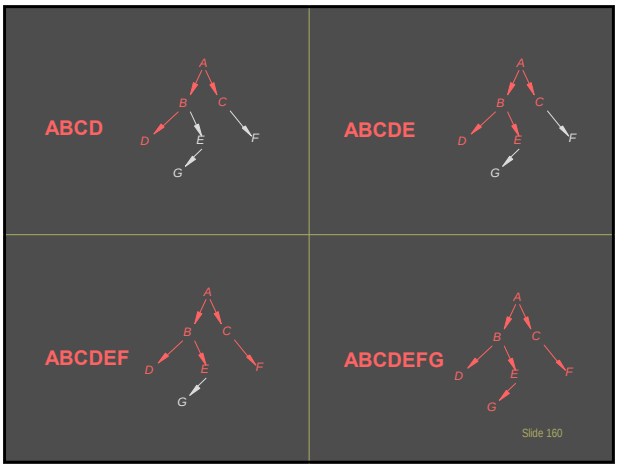
157


10.3.4 Breadth-First Traversal
ABCDEFG
KIT107 ©JRD, 2021
Slide 158

158


A
AB
ABC
Slide 159

159



160

10.3.5 Traversal Algorithms

- Traversal algorithms could be written in a straight-forward manner to emulate the previous diagrams, or
- Data Structures could be used to allow a generic algorithm to provide both depth-first and breadth-first traversal

161

DFT without Intermediate Data Structures

```
void traverseDF(bintree t)
{
        bintree l,r;

        if (! isEmpty(t))
        {
                l=getLeft(t);
                r=getRight(t);
                traverseDF(l);
                traverseDF(r);
                f(getRootData(t)); // post-order
        }
}
```

Whatever f() does…

162

BFT without Intermediate Data Structures

```
void traverseBF(bintree t)
{
      int i;

      for (i=1; i<=height(t);i++)
            processLevel(t,i);
}

int height(bintree t)
{
      …
}
```

See lab class

163

BFT without Intermediate Data Structures (continued)

```
void processLevel(bintree t,int n)
{
      bintree l,r;

      if (! isEmpty(t)) {
            if (n==1)     //terminating case
                  f(getRootData(t));
            else {        //recursive case
                  l=getLeft(t);
                  r=getRight(t);
                  processLevel(l,n-1);
                  processLevel(r,n-1);
            }
      }
}
```

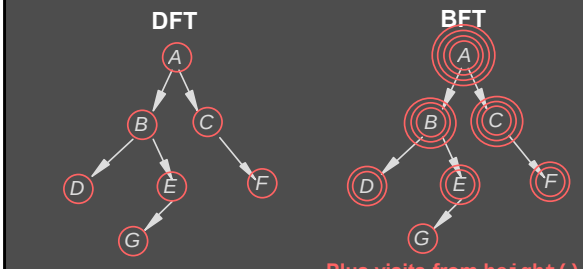
Whatever f() does…

BinTree.c

Slide172.sln

164

Comparison



Plus visits from height()

165

Comparison

- Both are special purpose (i.e. different) algorithms
- DFT has worst case of W(*n*)=*n*
- BFT has worst case of W(*n*) >> *n*
- Could a data structure replace specialised algorithms to improve performance?

166

Intermediate Data Structures

- For DFT the aim is to visit nodes on the current branch, i.e. we want to start again at the *most* recent node
- For BFT the aim is to process in levels and abandon the current branch, i.e. we want to start at the *least* recent node

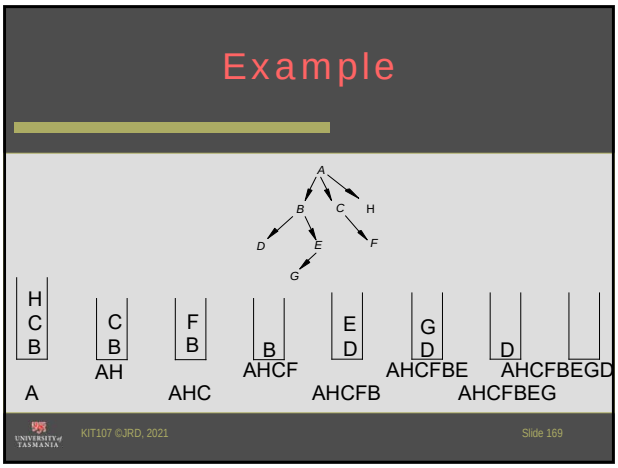
167

DFT with Intermediate Stack

- If root of tree is empty, stop
- If root of tree is stop condition, stop
- From the left-most child to the right-most child of the root, if the child is not empty, then push the child onto the stack
- If the stack is not empty then
  - pop the top item from the stack
  - recurse with the tree which is rooted at the popped item and the current stack

168

Example



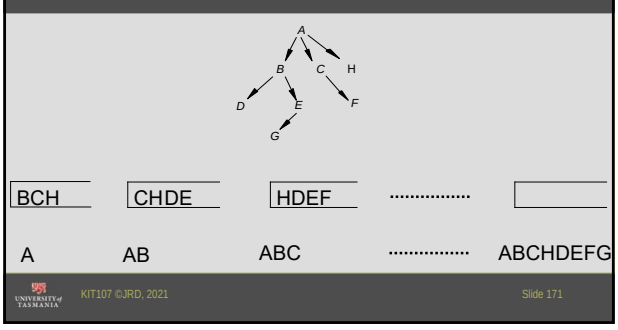
169

BFT with Intermediate Queue

- If root of tree is empty, stop
- If root of tree is stop condition, stop
- From the left-most child to the right-most child of the root, if the child is not empty, then add the child onto the queue
- If the queue is not empty then
  - remove the front item from the queue
  - recurse with the tree which is rooted at the removed item and the current queue

170

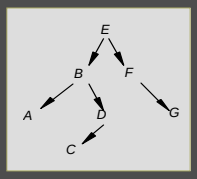
Example



171

10.4 Binary Search Trees

- *Binary search trees* are Binary Trees in which nodes are ordered:
  - all nodes in left sub-tree have values smaller than parent
  - all nodes in right sub-tree have values larger than parent



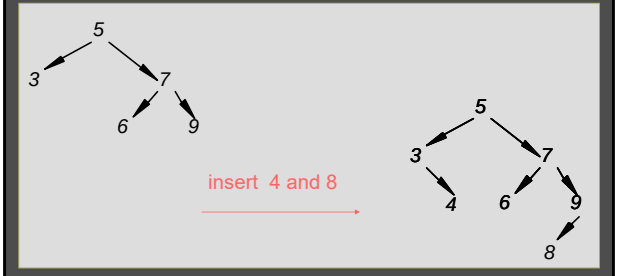
172

10.4.1 Insertion Algorithm

- Insertion occurs only at empty branches
- Begin at the root node
  - if the value to be added is smaller go left otherwise go right; repeat search
- When no further movement is possible
  - add value as new left child if smaller; add value as new right child if larger

173

Insertion Example



174

10.4.2 Deletion Algorithm (Overview)

- Deletion can occur anywhere in the tree
- If a leaf node is to be deleted it may simply be removed
- If a non-leaf node is to be deleted it can be replaced by its only branch, or if it has two branches by a node within the sub-tree

175

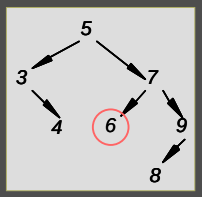
Deletion Algorithm

- Locate the value to be deleted by searching (as done in the Insertion Algorithm)
- If the value cannot be found report an error and stop
- If the node containing the value is a leaf node delete it and stop

176

Deletion Algorithm

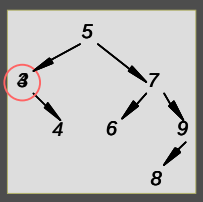
- If the node containing the value is a leaf node delete it and stop



177

Deletion Algorithm (continued)

- If the node has one branch replace the node with the branch



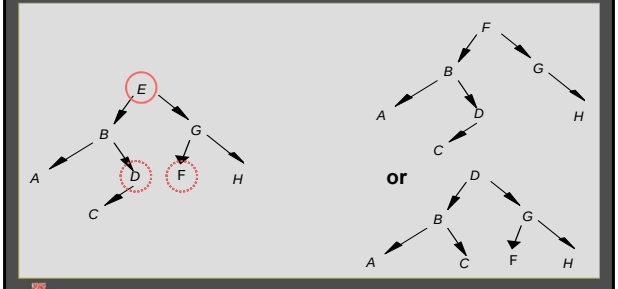
178

Deletion Algorithm (continued)

- If the node has one branch replace the node with the branch
- If the node has two branches
  - either replace the node with the right-subtree's left-most node (smallest value)
  - or replace the node with left-subtree's right-most node (largest value)
  - and delete the value from the sub-tree

179

Deletion Example



180

10.5 Search Trees in General

- A *search tree of degree n* is a tree where each node has a maximum of *n* branches
- Nodes may have up to *n*–1 values — a node with *k*–1 values has *k* branches (some of which may be empty)
- Node values are ordered…

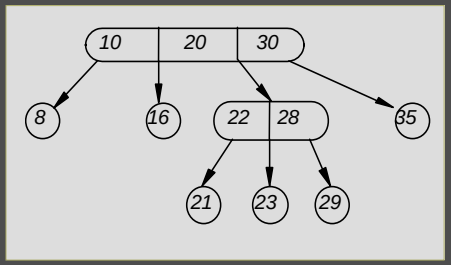
181

Search Trees in General (continued)

- Nodes containing *k* branches have *k*–1 values such that:
  - all values in the sub-tree of branch *i* have values *less than* value *i*
  - all values in the sub-tree of branch *i*+1 have values *greater than* value *i*

182

Example



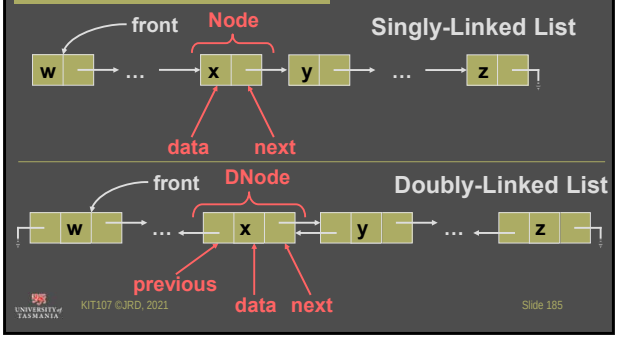
183

11. Doubly-Linked Lists

- 11.1 Concept
- 11.2 Implementation
- 11.3 Example: Stack

184

11.1 Concept



185

11.2 Implementation

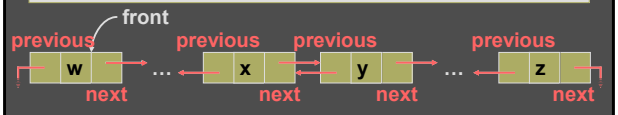
```
struct dnode_int {
     dnode prev;
     void *data;
     dnode next;
};
typedef struct dnode_int *dnode;
```

- A dnode is a triple of `prev`, `data`, and `next`
- `prev` refers to the preceding node just as `next` refers to the succeeding node

186

Implementation (continued)

```
struct dnode_int {
     btnode prev;
     void *data;
     btnode next;
};
typedef struct dnode_int *dnode;
```



187

11.3 Example: Stack

- Note: the use of a doubly-linked list as the underlying implementation for the Stack ADT does not utilise all the functionality of the doubly-linked list data structure

188

12. Variations on the Queue

- 12.1 Double-ended queue (deque)
- 12.2 Circular queue (`cirque`)
- 12.3 Priority queue (`prique`)

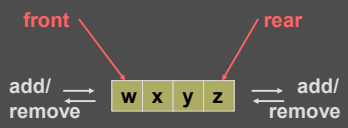
189

12.1 Double-Ended Queue

- A *double-ended queue* (deque) is a data structure with 'queues at both ends' — items may be added at either end and removed from either end, but only the ends are accessible
- The Deque ADT may be implemented in C using an array, a singly-linked list, or a doubly-linked list

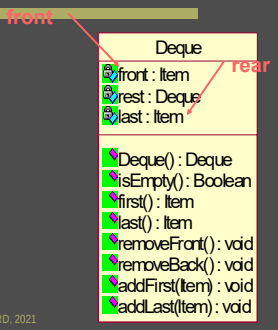
190

12.1 Double-Ended Queue



191

12.1.1 Deque UML Diagram



192

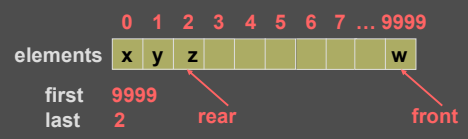
12.1.2 Deque Implementation

```
struct deque_int;
typedef struct deque_int *deque;

void init_deque(deque *d);
bool isEmpty(deque d);
void *first(deque d);
void *last(deque d);
void addFirst(deque d,void *o);
void addLast(deque d,void *o);
void removeFront(deque d);
void removeBack(deque d);
```

193

12.1.2 Deque Implementation

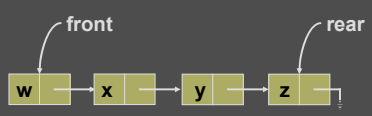
- Array — exercise!



194

12.1.3 Deque Implementation

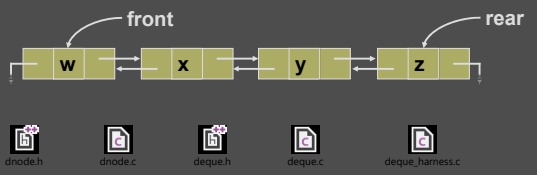
- Singly-linked list — exercise!



195

12.1.4 Deque Implementation

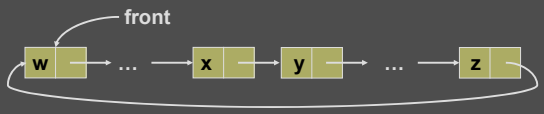
- Doubly-linked list — follows…



196

12.2 Circular Queues

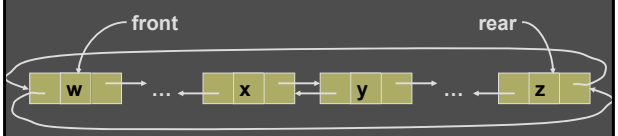
- A *circular queue* (cirque) is a queue in which the rear of the queue is connected to the front of the queue



197

12.2 Circular Queues (continued)

- A double-ended circular queue is a deque in which the rear of the queue is connected to the front, and the front of the is connected to the rear
- Double-ended circular queue using a doubly-linked list:



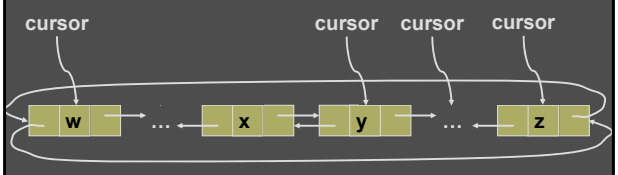
198

Circular Queues with a Cursor

- It is also possible to have a reference to the 'current' node
- This concept is called a *cursor* and is analogous to a computer screen's cursor

199

Circular Queues with a Cursor (continued)



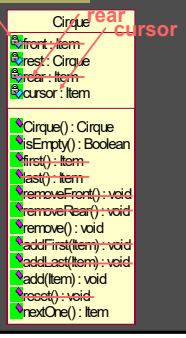
200

Circular Queues with a Cursor (continued)

- When a cursor is used, two additional routines are required
  - `nextOne()` which returns the current node and advances the cursor to the next node
  - `reset()` which sets the cursor back to be the first item in the circular queue
  - `add()` and `remove()` are also possible

201

12.2.1 Cirque UML Diagram



202

12.2.2–4 Cirque Implementation

- Doubly-linked list — see lab class
- Singly-linked list — exercise!
- Array — exercise!

203

12.3 Priority Queues

- Every ADT seen so far stores data in a place determined by the ADT and not the value of the data
- Search trees have been an exception
- Priority queues are another exception

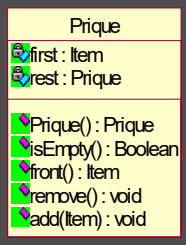
204

12.3 Priority Queues (continued)

- A *priority queue* (prique) is a queue in which the items are inserted into a place in the queue that is determined by that item's priority/importance/value
- Apart from changes to the `add()` operation, the `Prique` implementation is unchanged from the `Queue`

205

12.3.1 Prique UML Diagram



206

Comparing Apples and Oranges

- A comparison function must exist (or be provided) in order for arbitrary priorities to be compared
- This comparison function is needed within `add()`
- In order for it to be available it must be passed in on the parameter list

207

Comparing Apples and Oranges (continued)

```
void add(prique p, void *o,
         bool(*greaterThan)(void *,void *))
{
     …
     while ((c!=NULL) &&
          ((*greaterThan)(getData(c),o)))
     {
          …
     }
     …
}
```

208

12.3.2 Prique ADT Implementation

- Doubly-linked list — see lab class
- Singly-linked list — exercise!
- Array — exercise!

209

13. Cloning

- 13.1 Copying Variables

210

13.1 Copying Variables

- All of the ADTs implemented so far have been *destructive* — the `push()`, `pop()`, `add()`, `remove()`, `insert()`, `delete()`, ..., `union()`, `intersection()`, and `difference()` functions have all changed the contents of the passed variable

211

Destructive vs Constructive Implementations

- It is possible to write *constructive* implementations instead — a new dynamic variable is created by copying the original one, the modification is made to the new one, and then the new one is returned.
- How can you tell if an implementation is destructive/constructive?

212

The Queue ADT C Header

```
struct queue_int;
typedef struct queue_int *queue;

void init_queue(queue *q);
bool isEmpty(queue q);
queue add(queue q, void *i);
void *front(queue q);
queue rear(queue q);
```

213

Copying Dynamic Variables: Cloning

- Three ways to 'copy' a variable:
  - Copying the reference variable
  - Shallow cloning
  - Deep cloning

214

Copying Values

```
int a[]={10,20,30};

int *copy;
copy=a;

copy[0]=40;
printf("%d",a[0]);
```

- This only copies the pointer it doesn't copy the array

215

Copying Values

```
int a[]={10,20,30};
int *copy;
int i;

copy=(int *)malloc(3*sizeof(int));
for (i=0;i<3;i++) { copy[i]=a[i]; }

copy[0]=40;
printf("%d",a[0]);
```

- This copies the pointer and the array

216

Copying Values

```
char *a[]={"a","b","c"};
char **copy;
int i;

copy=(char **)malloc(3*sizeof(char *));
for (i=0;i<3;i++) { copy[i]=a[i]; }

copy[0]="x";
printf("%s",a[0]);
```

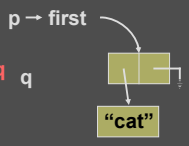
- This copies the pointer and the array, but not the pointees

217

Cloning (continued)

```
queue p,q;
char s[]={"cat"};

init_queue(&p);
add(p,s);
clone(p,&q);
```

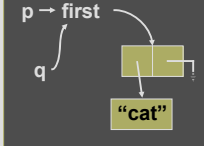
Copy p into q



218

Cloning (continued)

```
clone(p,&q);
-------------------------
q=p;
-------------------------
void clone(queue p,queue *qp)
{
     *qp=p;
}
```

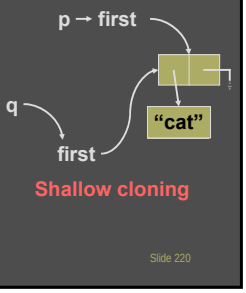


Copying the reference

219

Cloning (continued)

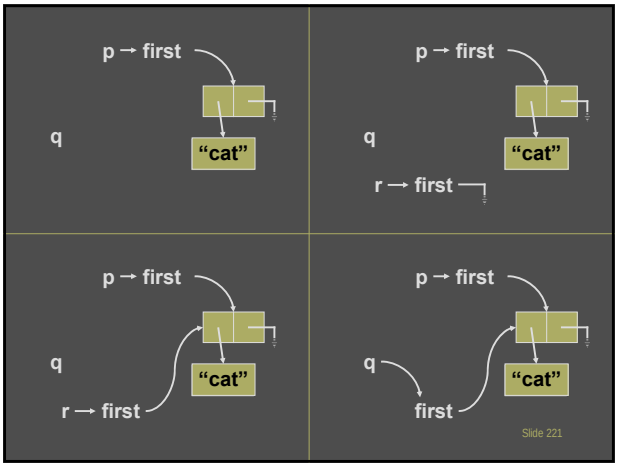
```
clone(p,&q);
-------------------------
q->first=p->first;
-------------------------
void clone(queue p,queue *qp)
{
     queue r;

     init_queue(&r);
     r->first=p->first;
     *qp=r;
}
```



Shallow cloning

Slide 220

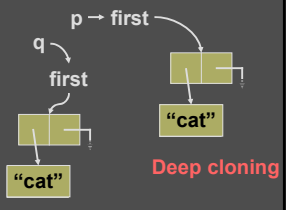
220



221

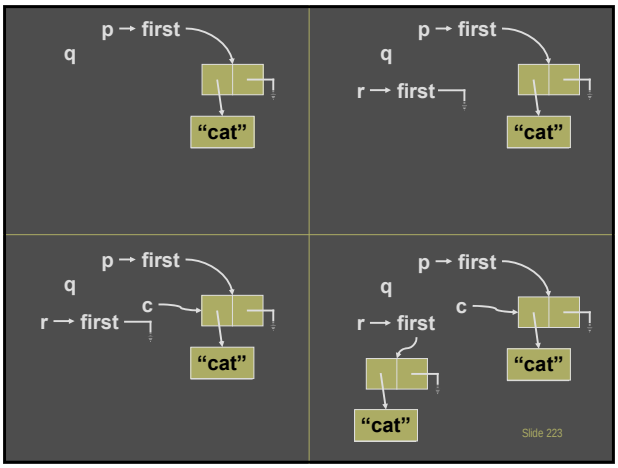
Cloning (continued)

```
clone(p,&q);
-------------------------
void clone(queue p,queue *qp)
{
  char *s;
  queue r;
  node c=p->first;

  init_queue(&r);
  while (c != NULL) {
    s=(char *)malloc(((strlen(getData(c))+1)*sizeof(char));
    strcpy(s,getData(c));
    add(r,s);
    c=getNext(c);
  }
  *qp=r;
}
```

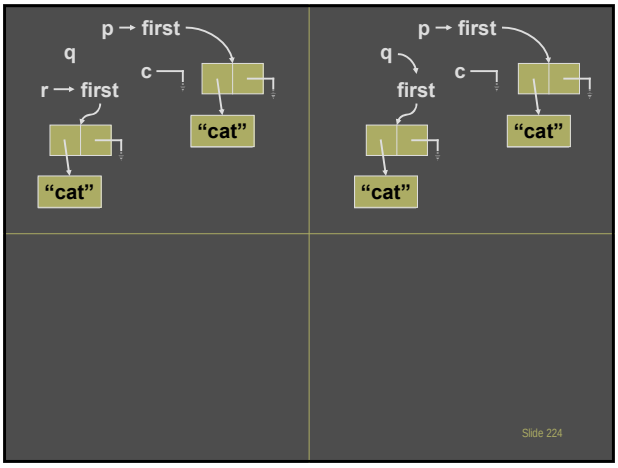


Deep cloning

222



223



224

How do we clone the contents?

- The data field is `(void *)`
- Q. How do we copy the pointee as well as the pointer?
- A. Pass a function to do it!
- (or use `memcpy()` from `<string.h>`)

225

Cloning (continued)

```
void clone(queue p,queue *qp,
         void(*cloneData)(void *,void **))
{
  void *o;
  queue r;
  node c=p->first;

  init_queue(&r);
  while (c != NULL) {
    (*cloneData)(getData(c),&o);
    add(r,o);
    c=getNext(c);
  }
  *qp=r;
}
```

226

The Queue ADT C Header

```
void init_queue(queue *q);
bool isEmpty(queue q);
queue add(queue q, void *i,
  void(*cloneData)(void *,void **));
void *front(queue q);
queue rear(queue q,
  void(*cloneData)(void *,void **));
void clone(queue p, queue *q,
  void(*cloneData)(void *,void **));
```

227

Exercise

- **Once we've seen the Set, re-write the implementation to provide a constructive rather than destructive implementation**

228

14. Algorithm Efficiency

- 14.1 Measuring Efficiency
- 14.2 Comparing Efficiency
- 14.3 Example

229

14.1 Measuring Efficiency

- A measure is required that describes the work done by (or memory usage of) an algorithm
- This measure is called *complexity*
- Given such a measure, comparisons of algorithms can be made so that efficient algorithms may be selected for use

230

Different Complexities

- Space Complexity
  - Memory consumed by algorithm
  - Measure: number of variables and count x size of dynamically created objects
- Time Complexity
  - Measure count of executions of a 'fundamental operation'

231

Example

Problem	Most important operation
Search a list	Comparisons with current list element and desired value
Sorting a list	Comparing pairs of list elements
Traverse a list	Traversing a reference

232

Describing Complexity

- Complexity depends upon the size of the input — small data-set requires little work, large data-set requires great work
- Let $T_A(n)$ stand for the time complexity of an algorithm *A* given input of size *n*

233

Example

- Assuming `getNext()` is the fundamental operation and that the list contains *n* nodes

```
0          c=front;
? x        while (c != NULL)
1             c=getNext(c);
```

the time complexity of this program fragment, $T(n)=0+?*1 =?*1 =n*1 =n$

234

Best, Worst, Average

- For all inputs of size *i*, 1 <= i <= *n*
  - The *best case* complexity for algorithm A is $B_A(n)$=minimum($T_A(i)$)
  - The *worst case* complexity is $W_A(n)$=maximum($T_A(i)$)
  - The *average case* complexity is $A_A(n)$=sum(p(i)$T_A(i)$)

  where input *i* occurs with probability p(*i*)

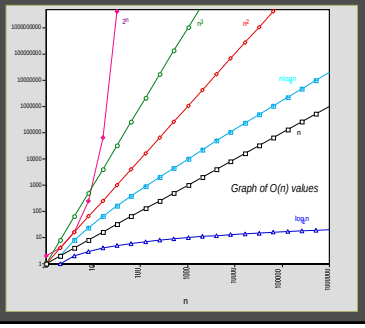
235

14.2 Comparing Efficiency

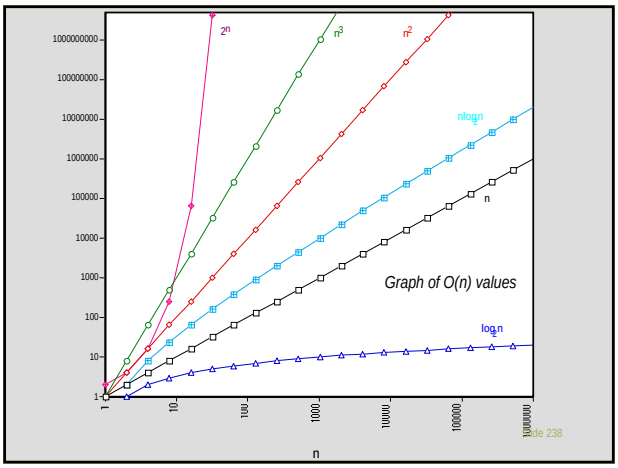
- Counts of fundamental operations are only indicative of work done
- Approximations of time complexities only are compared
- Algorithms are classified into groups/families based on the dominant term of the approximated time complexity

236

Some Typical 'Families'



237



238

Big-oh Notation

- Big-oh notation, O(), is used to indicate an approximation of algorithm complexity
- An algorithm's time complexity may be expressed in big-oh notation by keeping the term of highest exponent, discarding all other terms, and discarding the coefficient

239

Examples

$W(n) = 5n^3 + n^2 + n + 10$

This complexity is $O(n^3)$

$W(n) = 50 \ (= 50n^0 = 50x1)$

This complexity is O(1)

240

14.2.1 Definition of Big-oh

- Given functions f(n) and g(n), if there exist constants c and M such that: f(n) <= cg(n) for n >= M, where c > 0 and M > 0, then, f(n) is said to be O(g(n)).
- Example:
  - W(n) = $5n^3 + n$
    For M=1 and c=6, $5n^3 + n <= cn^3$ is true
    hence, $5n^3 + n$ is O(n^3)

241

14.3 Example

- Four algorithms (A, B, C, and D) perform the same task but possess different complexities
- The fundamental operation requires 0.000001s

Algorithm	n=1	n=50	n=100	n=1000	n=10000
$W_A(n)$ O(n)					
$W_B(n)$ O($\log_2 n$)					
$W_C(n)$ O(n^3)					
$W_D(n)$ O(2^n)					

242

15. Sets and Bags

- 15.1 Operations
- 15.2 UML Diagram
- 15.3 Implementation
- 15.4 Sample Harness
- 15.5 Bags

243

15.1 Set Operations

- A *set* is an unordered collection of zero or more elements in which no element occurs more than once
- A *bag* is an unordered collection of zero or more elements in which elements may occur more than once
- The empty set/bag is written {} or sometimes ∅

244

Set Operations (continued)

- *elementOf* (∈) — check whether a value is in a set, e.g. 5 ∈ {3,1,5,2} is true and 7 ∈ {3,1,5,2} is false
- *cardinality* (||) — calculate the number of elements in a set, e.g. |{}| = 0 and |{3,1,5,2}| = 4

245

Set Operations (continued)

- *union* (∪) — the union of two sets A and B is the set of elements in A or B, e.g. {3,1,5,2} ∪ {5,6,1,3} = {2,5,1,6,3}
- *intersection* (∩) — the intersection of two sets A and B is the set of elements in both A and B, e.g. {3,1,5,2} ∩ {5,6,1,3} = {1,5,3}

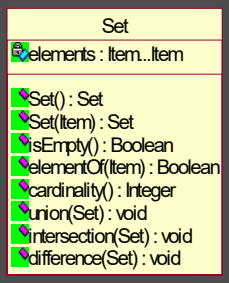
246

Set Operations (continued)

- ***difference*** **(\ or –) — the difference of a set A from a set B is the set of all elements in A that are not in B, e.g. {3,1,5,2} \ {5,6,1,3} = {2}**

247

15.2 The Set UML Diagram



248

15.3 Set Implementations

- **There are many alternatives including:**
  - **a linked list of `(void *)` values**
    - **exercise!**
  - **an array of `(void *)` values (the total number of potential elements must be known)**
    - **exercise!**

249

Set Implementations (continued)

- and
  - an array of `bool` where each set element has a corresponding array element that indicates whether the set element is in the array (`true`) or not (`false`) (the total number of potential elements must be known)
    - following…
    - Consider the set of possible colours {RED, BLUE, YELLOW, PURPLE, GREEN, ORANGE}

250

Example

```
typedef enum {RED=0, BLUE=1, YELLOW=2,
     PURPLE=3, GREEN=4, ORANGE=5}
     colours;

bool elements[];
```

	RED	BLUE	YELLOW	PURPLE	GREEN	ORANGE
elements	false	true	true	false	false	false

251

Set ADT Header File

```
struct set_int;
typedef struct set_int *set;

void init_set(set *sp,int n,bool x,
                  int e);
bool isEmpty(set s);
void setUnion(set s1,set s2);
void intersection(set s1,set s2);
void difference(set s1,set s2);
bool elementOf(set s,int e);
int cardinality(set s);
```

252

15.4 Set Implementation

253

15.5 Bags

- Bags have similar operations to sets
- Bag operations use addition/subtraction rather than checking/defining the presence of elements
- An additional operation is required:
  - `count(Item) : int` — this determines the number of times a particular element is present in a bag

254

Exercise

- Modify the implementation of the Set ADT to provide a definition and implementation of the Bag

255

16. Bitwise Operations

16.1 Operators
16.2 Example

256

16.1 Operators

- & — bit-wise *and*
- | — bit-wise *or*
- ^ — bit-wise *exclusive or* (xor)
- ~ — bit-wise *complement* (twiddle)
- << — left shift
- >> — right shift

257

Bit-wise and

- `x & y` sets the bits whenever the same bits are set in each of x and y

```
2 & 6 is 2
4 & 9 is 0
```


258

Bit-wise or

- x | y sets the bits whenever the same bits are set in either or both of x and y

  2 | 6 is 6
  4 | 9 is 13

259

Bit-wise xor

- x ^ y sets the bits whenever the same bits are set in either (*but not both*) of x and y

  2 ^ 6 is 4
  4 ^ 9 is 13

260

Bit-wise complement

- ~x swaps all the bits in x so that all 0s become 1s and vice-versa

261

Left shift

- x << y shifts all the bits in x to the left by y bits, i.e. multiply by 2^y

  2 << 6 is 128, i.e. 2 * 64
  9 << 1 is 18, i.e. 9 * 2

262

Right shift

- x >> y shifts all the bits in x to the right by y bits, i.e. divide by 2^y

  135 >> 3 is 16, i.e. 135 / 8
  9 >> 1 is 4, i.e. 9 / 2

263

16.2 Example

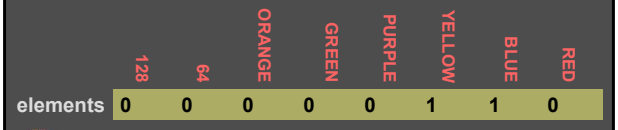
- Revisit the set to provide a more space-efficient implementation

264

Example

```
typedef enum {RED=1, BLUE=2, YELLOW=4,
     PURPLE=8, GREEN=16, ORANGE=32}
     colours;

char elements;
```

Only 8 bits (1 byte!)



265

Set ADT Header File

```
struct set_int;
typedef struct set_int *set;

void init_set(set *s,bool x,int e);
bool isEmpty(set s);
void setUnion(set s1,set s2);
void intersection(set s1,set s2);
void difference(set s1,set s2);
bool elementOf(set s,int e);
int cardinality(set s);
```

266

Implementation

267

Problem

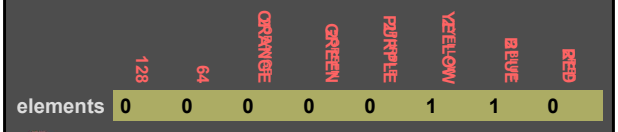
```
typedef enum {RED=1, BLUE=2, YELLOW=4,
     PURPLE=8, GREEN=16, ORANGE=32}
     colours;
```

- The numbers for the values give away the implementation
- The data structure shouldn't be apparent in the ADT

268

Example

```
typedef enum {RED, BLUE, YELLOW,
     PURPLE, GREEN, ORANGE}
     colours;

char elements;
```



269

Implementation

270

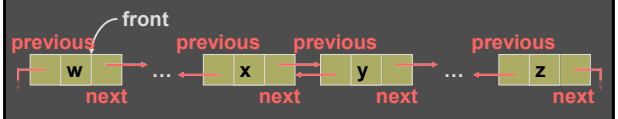
17. A Java Priority Queue

* Not examinable! Just for interest

271

17.1 Linked Lists

- A dnode is a triple of `prev`, `data`, and `next`
- `prev` refers to the preceding node just as `next` refers to the succeeding node

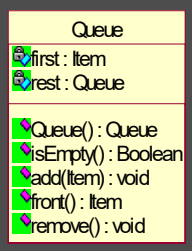


Interface Class

272

17.2 Genericity

- The `Object` class represents any item
- Example: Queue ADT



Interface Class Driver

273

17.3 Cloning

```
Queue p,q;
String s = "cat";

p = new Queue();
p.add(s);
q = p.clone();
```


274

Cloning (continued)

- Convention is to write a method called `clone()`
- To advertise that this exists, we implement the `Cloneable` interface

Interface Class

275

17.4 Constructive Implementations

```
public interface QueueInterface
{
    //public Queue();
    //public Queue(Object o);
    public boolean isEmpty();
    public Object front() throws
          EmptyQueueException;
    public Queue add(Object o);
    public Queue remove() throws
          EmptyQueueException;
}
```

Class

276

17.5 Generics ('Templates')

- Using `Object` brings two problems that we identified earlier:
  - consistency of type content and
  - necessity for type casting
- If we could specify what 'kind' of `Objects` we would accept, that would solve the problems — we can using 'templates'

277

Generics (continued)

- *Generics* are a kind of combination between:
  - giving the class definition a 'type parameter' and
  - using a pre-processor to 're-write' the program substituting the type parameter for `Object`

278

Generics (continued)

Type parameter

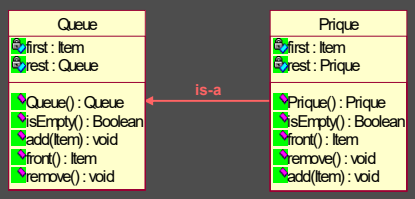
```
public interface QueueInterface<T>
{
    //public Queue();
    //public Queue(T o);
    public boolean isEmpty();
    public T front() throws
          EmptyQueueException;
    public Queue<T> add(T o);
    public Queue<T> remove() throws
          EmptyQueueException;
}
```

Class

279

17.6 Inheritance

- A priority queue is a queue with special behaviour for add( )



280

Inheritance (continued)

```
public class Prique<T> extends Queue<T>
     implements PriqueInterface<T>, Cloneable
{
    public Prique<T> add(T o)
    {
        …
    }
}
```

- Everything else is inherited!

281

Comparing Apples and Oranges

- add( ) needs a comparison method so that arbitrary types of values can be compared
- We can use type polymorphism to help by restricting the kinds of values we want to just those which are known to be comparable

282

Comparable Interface

- Like Cloneable, there exists an interface called Comparable which can be implemented to indicate values are able to be compared
- The interface specifies only one method: compareTo()
- We need to restrict our implementation to only accept Comparable types

283

Comparable Interface (continued)

```
public class Prique <T extends Comparable<T>>
     extends Queue<T>
     implements PriqueInterface<T>, Cloneable
{
    …
}
```

Interface Class Driver

284