ALevel CS C23 Algorithm (1)

Binary search: repeated checking of the middle item in an ordered search list and discarding the half of the list which does not contain the search item

```
Found ← FALSE
SearchFailed ← FALSE
First ← 0
Last ← MaxItems – 1 // set boundaries of search area
WHILE NOT Found AND NOT SearchFailed DO
   Middle ← (First + Last) DIV 2 // find middle of current search
area
   IF List[Middle] = SearchItem
      THEN
         Found ← TRUE
      ELSE
         IF First >= Last // no search area left
            THEN
               SearchFailed ← TRUE
            ELSE
               IF List[Middle] > SearchItem
                  THEN // must be in first half
                     Last ← Middle - 1 // move upper boundary
                  ELSE // must be in second half
                     First ← Middle + 1 // move lower boundary
               ENDIF
         ENDIF
   ENDIF
ENDWHILE
IF Found = TRUE
   THEN
      OUTPUT Middle // output position where item was found
   ELSE
      OUTPUT "Item not present in array"
ENDIF
```

Linear search:

```
MaxIndex ← 6
INPUT SearchValue
Found ← FALSE
Index ← –1
REPEAT
   Index ← Index + 1
   IF MyList[Index] = SearchValue
      THEN
         Found ← TRUE
   ENDIF
UNTIL FOUND = TRUE OR Index >= MaxIndex
IF Found = TRUE
   THEN
      OUTPUT "Value found at location: " Index
   ELSE
      OUTPUT "Value not found"
ENDIF
```

Bubble sort:

```
n ← MaxIndex – 1
FOR i ← 0 TO MaxIndex – 1
   FOR j ← 0 TO n
      IF MyList[j] > MyList[j + 1]
         THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
      ENDIF
   NEXT j
   n ← n – 1 // this means the next time
round the inner loop, we don't
          // look at the values already in
the correct positions.
NEXT i
```

Insertion sort:

```
FOR Pointer ← 1 TO NumberOfitems – 1
   ItemToBeInserted ← List[Pointer]
   CurrentItem ← Pointer – 1 // pointer to last item in sorted part of list
   WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem > –1) DO
      List[CurrentItem + 1] ← List[CurrentItem] // move current item down
      CurrentItem ← CurrentItem – 1 // look at the item above
   ENDWHILE
   List[CurrentItem + 1] ← ItemToBeInserted // insert item
NEXT Pointer
```

Create Linked List:

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointer
TYPE ListNode
   DECLARE Data : STRING
   DECLARE Pointer : INTEGER
ENDTYPE Su
DECLARE StartPointer : INTEGER
DECLARE FreeListPtr : INTEGER
DECLARE List : ARRAY[0 : 6] OF ListNode

PROCEDURE InitialiseList
   StartPointer ← NullPointer // set start pointer
   FreeListPtr ← 0 // set starting position of free list
   FOR Index ← 0 TO 5 // link all nodes to make free list
      List[Index].Pointer ← Index + 1
   NEXT Index
   List[6].Pointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Delete a node from an ordered linked list

```
PROCEDURE DeleteNode(DataItem)
   ThisNodePtr ← StartPointer // start at beginning of list
   WHILE ThisNodePtr <> NullPointer // while not end of list
    AND List[ThisNodePtr].Data <> DataItem DO // and item not found
      PreviousNodePtr ← ThisNodePtr // remember this node
      // follow the pointer to the next node
      ThisNodePtr ← List[ThisNodePtr].Pointer
   ENDWHILE
   IF ThisNodePtr <> NullPointer // node exists in list
      THEN
         IF ThisNodePtr = StartPointer // first node to be deleted
            THEN
               // move start pointer to the next node in list
               StartPointer ← List[StartPointer].Pointer
            ELSE
               // it is not the start node;
               // so make the previous node's pointer point to
               // the current node's 'next' pointer; thereby removing all
               // references to the current pointer from the list
               List[PreviousNodePtr].Pointer ← List[ThisNodePtr].Pointer
         ENDIF
         List[ThisNodePtr].Pointer ← FreeListPtr
         FreeListPtr ← ThisNodePtr
   ENDIF
ENDPROCEDURE
```

Insert a node into an ordered list:

```
PROCEDURE InsertNode(NewItem)
   IF FreeListPtr <> NullPointer
      THEN // there is space in the array
         // take node from free list and store data item
         NewNodePtr ← FreeListPtr
         List[NewNodePtr].Data ← NewItem
         FreeListPtr ← List[FreeListPtr].Pointer
         // find insertion point
         ThisNodePtr ← StartPointer // start at beginning of list
         PreviousNodePtr ← NullPointer
         WHILE ThisNodePtr <> NullPointer // while not end of list
            AND List[ThisNodePtr].Data < NewItem DO
               PreviousNodePtr ← ThisNodePtr // remember this node
               // follow the pointer to the next node
               ThisNodePtr ← List[ThisNodePtr].Pointer
         ENDWHILE
         IF PreviousNodePtr = StartPointer
            THEN // insert new node at start of list
               List[NewNodePtr].Pointer ← StartPointer
               StartPointer ← NewNodePtr
            ELSE // insert new node between previous node and this node
               List[NewNodePtr].Pointer ← List[PreviousNodePtr].Pointer
               List[PreviousNodePtr].Pointer ← NewNodePtr
         ENDIF
   ENDIF
ENDPROCEDURE
```

Find an element in an ordered linked list

```
FUNCTION FindNode(DataItem) RETURNS INTEGER // returns pointer to node
   CurrentNodePtr ← StartPointer // start at beginning of list
   WHILE CurrentNodePtr <> NullPointer // not end of list
     AND List[CurrentNodePtr].Data <> DataItem DO // item not found
      // follow the pointer to the next node
      CurrentNodePtr ← List[CurrentNodePtr].Pointer
      ENDWHILE
      RETURN CurrentNodePtr // returns NullPointer if item not found
ENDFUNCTION
```

Access all nodes stored in the linked list

```
PROCEDURE OutputAllNodes
   CurrentNodePtr ← StartPointer // start at beginning of list
   WHILE CurrentNodePtr <> NullPointer DO // while not end of list
      OUTPUT List[CurrentNodePtr].Data
      // follow the pointer to the next node
      CurrentNodePtr ← List[CurrentNodePtr].Pointer
   ENDWHILE
ENDPROCEDURE
```

ALevel CS C23 Algorithm (2)

Create a new binary tree

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointers
TYPE TreeNode
   DECLARE Data : STRING
   DECLARE LeftPointer : INTEGER
   DECLARE RightPointer : INTEGER
ENDTYPE
DECLARE RootPointer : INTEGER
DECLARE FreePtr : INTEGER
DECLARE Tree : ARRAY[0 : 6] OF TreeNode
PROCEDURE InitialiseTree
   RootPointer ← NullPointer // set start pointer
   FreePtr ← 0 // set starting position of free list
   FOR Index ← 0 TO 5 // link all nodes to make free list
      Tree[Index].LeftPointer ← Index + 1
   NEXT Index
   Tree[6].LeftPointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Find a node in a binary tree

```
FUNCTION FindNode(SearchItem) RETURNS INTEGER // returns pointer to node
   ThisNodePtr ← RootPointer // start at the root of the tree
   WHILE ThisNodePtr <> NullPointer // while a pointer to follow
    AND Tree[ThisNodePtr].Data <> SearchItem DO // and search item not found
      IF Tree[ThisNodePtr].Data > SearchItem
         THEN // follow left pointer
            ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
         ELSE // follow right pointer
            ThisNodePtr ← Tree[ThisNodePtr].RightPointer
      ENDIF
   ENDWHILE
   RETURN ThisNodePtr // will return null pointer if search item not found
ENDFUNCTION
```

Insert a new node into a binary tree

```
PROCEDURE InsertNode(NewItem)
   IF FreePtr <> NullPointer
      THEN // there is space in the array
         // take node from free list, store data item, set null pointers
         NewNodePtr ← FreePtr
         FreePtr ← Tree[FreePtr].LeftPointer
         Tree[NewNodePtr].Data ← NewItem
         Tree[NewNodePtr].LeftPointer ← NullPointer
         Tree[NewNodePtr].RightPointer ← NullPointer
         // check if empty tree
         IF RootPointer = NullPointer
            THEN // insert new node at root
               RootPointer ← NewNodePtr
            ELSE // find insertion point
               ThisNodePtr ← RootPointer // start at the root of the tree
               WHILE ThisNodePtr <> NullPointer DO // while not a leaf node
                  PreviousNodePtr ← ThisNodePtr // remember this node
                  IF Tree[ThisNodePtr].Data > NewItem
                     THEN // follow left pointer
                        TurnedLeft ← TRUE
                        ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                     ELSE // follow right pointer
                        TurnedLeft ← FALSE
                        ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                  ENDIF
               ENDWHILE
               IF TurnedLeft = TRUE
                  THEN
                     Tree[PreviousNodePtr].LeftPointer ← NewNodePtr
                  ELSE
                     Tree[PreviousNodePtr].RightPointer ← NewNodePtr
               ENDIF
         ENDIF
   ENDIF
ENDPROCEDURE
```

Create a new queue

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = -1
CONSTANT MaxQueueSize = 8
DECLARE FrontOfQueuePointer : INTEGER
DECLARE EndOfQueuePointer : INTEGER
DECLARE NumberInQueue : INTEGER
DECLARE Queue : ARRAY[0 : MaxQueueSize – 1] OF STRING
PROCEDURE InitialiseQueue
   FrontOfQueuePointer ← NullPointer // set front of queue pointer
   EndOfQueuePointer ← NullPointer   // set end of queue pointer
   NumberInQueue ← 0   // no elements in queue
ENDPROCEDURE
```

Create a new stack

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = –1
CONSTANT MaxStackSize = 8
DECLARE BaseOfStackPointer : INTEGER
DECLARE TopOfStackPointer : INTEGER
DECLARE Stack : ARRAY[1 : MaxStackSize – 1] OF STRING

PROCEDURE InitialiseStack
   BaseOfStackPointer ← 0  // set base of stack pointer
   TopOfStackPointer ← NullPointer // set top of stack pointer
ENDPROCEDURE
```

Add an item to the queue

```
PROCEDURE AddToQueue(NewItem)
   IF NumberInQueue < MaxQueueSize
      THEN // there is space in the queue
         // increment end of queue pointer
         EndOfQueuePointer ← EndOfQueuePointer + 1
         // check for wrap-round
         IF EndOfQueuePointer > MaxQueueSize – 1
            THEN // wrap to beginning of array
               EndOfQueuePointer ← 0 // add item to end of queue
         ENDIF
         Queue[EndOfQueuePointer] ← NewItem
         // increment counter
         NumberInQueue ← NumberInQueue + 1
   ENDIF
ENDPROCEDURE
```

Pop an item off the stack

```
FUNCTION Pop()
   DECLARE Item : STRING
   Item ← EMPTYSTRING
   IF TopOfStackPointer > NullPointer
      THEN // there is at least one item on the stack
         // pop item off the top of the stack
         Item ← Stack[TopOfStackPointer]
         // decrement top of stack pointer
         TopOfStackPointer ← TopOfStackPointer – 1
   ENDIF
   RETURN Item
ENDFUNCTION
```

Remove an item from the queue

```
FUNCTION RemoveFromQueue()
   DECLARE Item : STRING
   Item ← EMPTYSTRING
   IF NumberInQueue > 0
      THEN // there is at least one item in the queue
         // remove item from the front of the queue
         Item ← Queue[FrontOfQueuePointer]
         NumberInQueue ← NumberInQueue – 1
         IF NumberInQueue = 0
            THEN // if queue empty, reset pointers
               CALL InitialiseQueue
            ELSE // increment front of queue pointer
               FrontOfQueuePointer ← FrontOfQueuePointer + 1
               // check for wrap-round
               IF FrontOfQueuePointer > MaxQueueSize – 1
                  THEN // wrap to beginning of array
                     FrontOfQueuePointer ← 0
               ENDIF
         ENDIF
   ENDIF
   RETURN Item
ENDFUNCTION
```

Push an item onto the stack

```
PROCEDURE Push(NewItem)
   IF TopOfStackPointer < MaxStackSize – 1
      THEN // there is space on the stack
         // increment top of stack pointer
         TopOfStackPointer ← TopOfStackPointer + 1
         // add item to top of
         Stack[TopOfStackPointer] ← NewItem
   ENDIF
ENDPROCEDURE
```

Hash table

hashing: Calculating an address from a key is called '**hashing**'.

If two different key values hash to the same address this is called a '**collision**'.

There are different ways to handle collisions:

1. chaining: create a linked list for collisions with start pointer at the hashed address
2. using overflow areas: all collisions are stored in a separate overflow area, known as 'closed hashing'
3. using neighbouring slots: perform a linear search from the hashed address to find an empty slot, known as 'open hashing'.

Insert a record into a hash table

```
PROCEDURE Insert(NewRecord)
   Index ← Hash(NewRecord.Key)
   WHILE HashTable[Index] NOT empty DO
      Index ← Index + 1 // go to next slot to check if empty
      IF Index > Max // beyond table boundary?
         THEN // wrap around to beginning of table
            Index ← 0
      ENDIF
   ENDWHILE
   HashTable[Index] ← NewRecord
ENDPROCEDURE
```

Find a record in a hash table

```
FUNCTION FindRecord(SearchKey) RETURNS Record
   Index ← Hash(SearchKey)
   WHILE (HashTable[Index].Key <> SearchKey)
    AND (HashTable[Index] NOT empty) DO
      Index ← Index + 1 // go to next slot
      IF Index > Max // beyond table boundary?
         THEN // wrap around to beginning of table
            Index ← 0
      ENDIF
   ENDWHILE
   IF HashTable[Index] NOT empty // if record found
      THEN
         RETURN HashTable[Index] // return the record
   ENDIF
ENDFUNCTION
```