Chapter 14 Linked Lists

Contents

Syllabus Learning Outcomes

1.3 Data Structures

Understand concept and write algorithms for stack, queue (linear and circular), linear linked list and binary search tree.

1.3.1 Understand the concept of static allocation of memory.

1.3.2 Understand the concept of dynamic allocation of memory.

1.3.4 Understand the concept of free space list (which could be another linked list or an array).

1.3.5 Create, update (edit, insert, delete) and search operations for a linear linked list.

Exclude: doubly-linked list and circular linked list

2.3 Implementing Algorithms and Data Structures

Use programming language elements and constructs to implement sort and search algorithms such as insertion sort, bubble sort, quicksort, merge sort, linear search, binary search and hash table search, as well as data structures such as stacks, queues, linear linked lists and binary search trees.

2.3.3 Write programs to implement operations for stacks, queues (linear and circular), linear linked lists and binary search trees.

Exclude: doubly-linked list and circular linked list

1 Abstract Data Type (ADT)

An **Abstract Data Type (ADT)** is a collection of data and a set of associated operations:

- create a new instance of the data structure
- insert a new element into the data structure
- delete an element from the data structure
- find/update an element in the data structure
- access all elements stored in the data structure in a systematic manner.

One can use an ADT's operations without knowing how the operations are implemented or how the data is stored. Examples of ADTs: Linked lists, stacks, queues and binary trees.

2 Linked Lists

An **array** is defined as a collection of items that are stored at **contiguous** (adjacent) memory locations. It is a container which can hold a **fixed number of items**, and these items should be of the **same type**.

A linked list is a **dynamic data structure**, which holds a collection of elements. The individual element may not be stored in contiguous memory locations but at whatever location that is available, and the elements are linked into an ordered sequence using pointers.

An element in the list is called a **node**. A node contains both data and a "link" to the next element. The "link" is usually referred to as a **pointer**, which is a variable that stores the address of the next node it points to.

A **pointer** that does not point at anything, indicating no further elements, is called a **null pointer**. It is usually represented by Ø. In program code, we set it to **None**. The **start/head** pointer stores the address of the first element (a link that points to the first node).

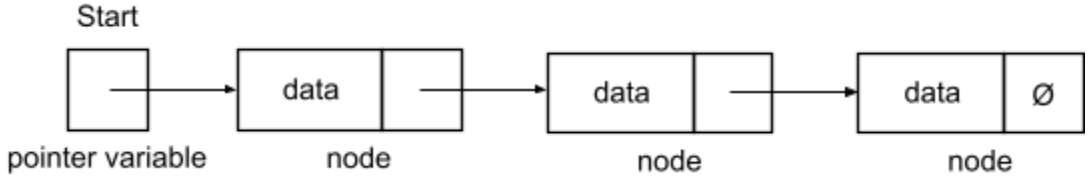
There are different types of linked list: singly linked list, doubly linked list and circular linked list.

A **singly linked list** is a collection of nodes, each containing one pointer pointing to the next node. The `start` pointer points to the first node, and the `pointer` in the last node points to null. When the list is empty, the `start` pointer points to null. With only one pointer, it can only traverse forward along the list.

A **doubly linked list** is a collection of nodes, each containing two pointers, one pointing to the previous node and the other pointing to the next node respectively. With two pointers, we can traverse forward and backward along the list. (Not in syllabus)

A **circular linked list** is similar to a singly linked list, but with the pointer in the last node pointing back to the first node. (Not in syllabus)

For the remaining of this chapter, we will focus on singly linked list.

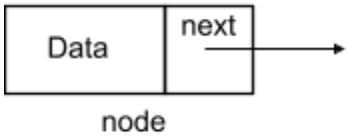


Conceptual diagram of a singly linked list

3 Operations of Singly Linked Lists using OOP

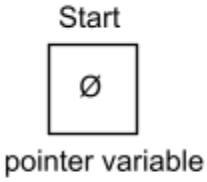
3.1 Creating a new linked list

In the examples which follow, we will assume each node consists of a `data` and a `next` pointer, pointing to the next node in the list.



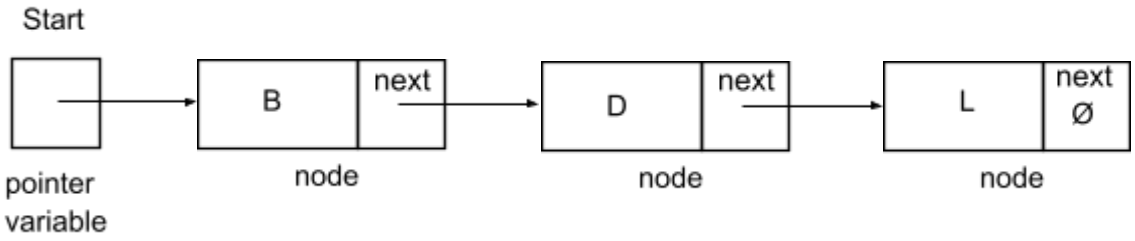
A class Node contains two attributes: `data` and a `next` pointer.

```
1  class Node:
2
3      def __init__(self, data):
4          """ constructor to initialize new node """
5          self.data = data
6          self.next = None
```



A class LinkedList contains one attribute: `start`. The constructor will initialise an empty linked list and set the `start` pointer to null.

```
11 class LinkedList:
12     """ collection of Node objects """
13     def __init__(self):
14         """ constructor to initialize empty linked list """
15         self.start = None
```

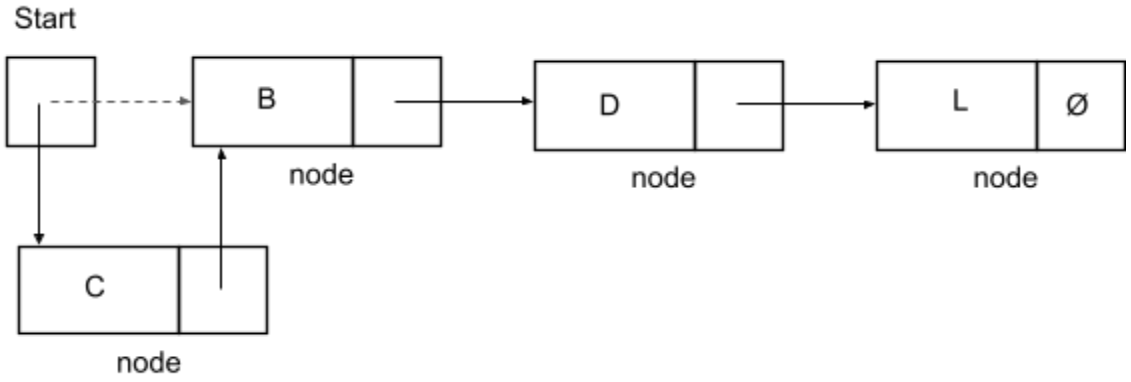


Conceptual diagram of a linked list

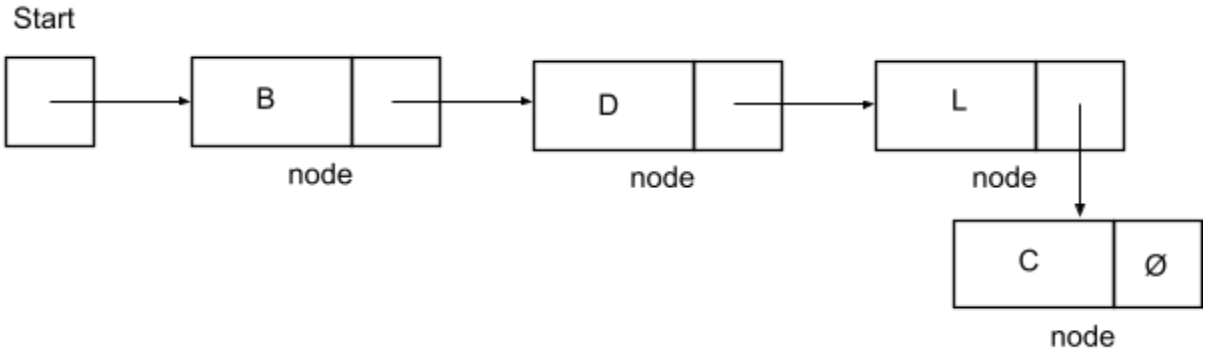
3.2 Inserting a node

A linked list can be either an ordered or unordered linked list. For an unordered linked list, a new node can be inserted either at the beginning of the list or at the end of the list.

To insert a new node, C, at the beginning of the list:
Step 1: the pointer in the new node, C, has to point to the content of the `Start` pointer
Step 2: the `Start` pointer is then set to point to the new node, C.

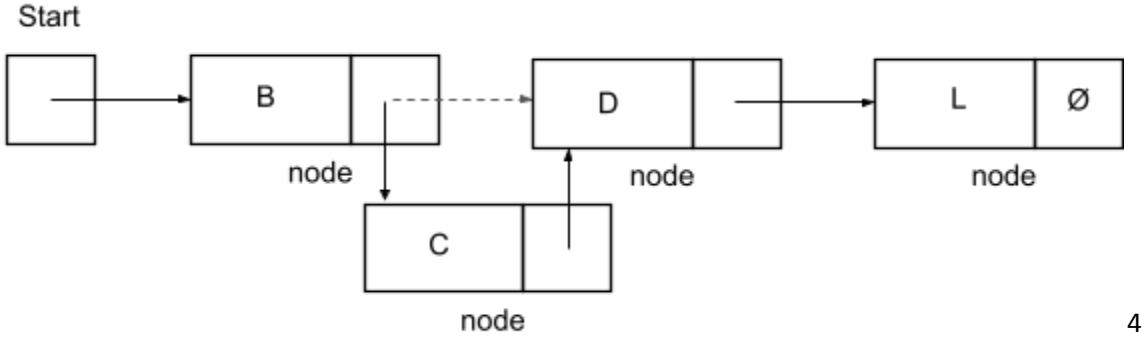


To insert the new note, C, at the end of the list:
Step 1: the pointer of node L (the last node) points to the new node, C. The pointer of the new node, C, contains the null pointer.



To insert a new node into an ordered linked list, the new node may need to be inserted between two existing nodes.

To insert a new node, C, between existing nodes, B and D:
Step 1: the pointer of node C has to point to the content of the pointer in B
Step 2: the pointer in B is then set to point to the new node, C.

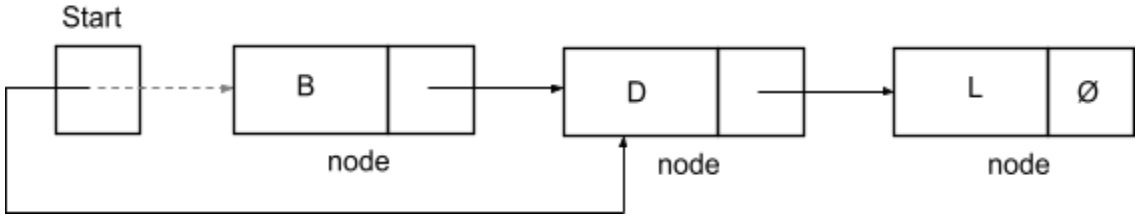


The algorithm for inserting a node into an ordered linked list:

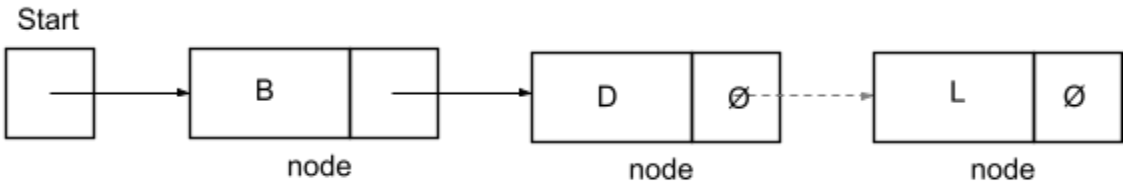
```
PROCEDURE insert(self, data)
  //Create a node object
  new_node  Node(data)

  IF self.start IS NONE  //Linked list is empty
    THEN
      self.start  new_node  //insert new_node as first node
    ELSE   //Find insertion point
      current  self.start  //start at beginning of list
      previous  current
      WHILE current IS NOT NONE AND current.data < data
        previous  current  //remember current node
        current  current.next  //follow the pointer to the next
node
      ENDWHILE
      IF previous == current  //insert new_node at the start of list
        THEN
          new_node.next  current
          self.start  new_node
        ELSE
          IF current IS NONE  //insert new_node at the end of list
            THEN
                previous.next  new_node
            ELSE  //insert new_node between previous and current
                new_node.next  current
                previous.next  new_node
          ENDIF
      ENDIF
  ENDIF
ENDPROCEDURE
```

3.3 Deleting a node

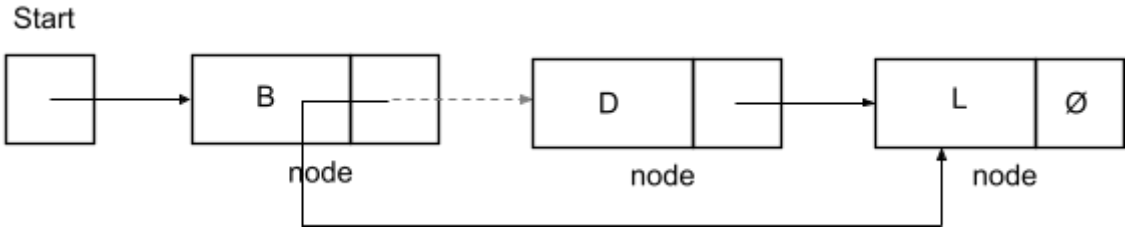
To delete the first node in the list, the `Start` pointer is set to point to the content of the pointer of the first node.



To delete the last node in the list, we set the pointer of the node before the last node to be null pointer.



To delete a node, D, within the list, we set the pointer of node B to point to the content of the pointer in D.



The algorithm for deleting a node from an ordered linked list:

```
PROCEDURE delete(self, data)
  IF self.start IS NONE
    THEN
      OUTPUT "Empty linked list."
    ELSE
      current  self.start
      previous  current
      WHILE current IS NOT NONE AND current.data < data
        previous  current
        current  current.next
      ENDWHILE
      IF current IS NONE or current.data <> data  //if data does not
                                                          exist
        THEN
          OUTPUT "Data does not exist!"
        ELSE  //data exists in current node
          IF current == self.start  //If data exists in first node
            THEN
              self.start  current.next  //delete first node
            ELSE
              previous.next  current.next  //delete current node
          ENDIF
      ENDIF
  ENDIF
ENDPROCEDURE
```

3.4 Search for data

With the `next` pointer, we can traverse down a list to search for data.

The algorithm for search in an ordered linked list:

```
FUNCTION search(self, data)
   IF self.start IS NONE  //Empty linked list
      THEN
           OUTPUT "Empty linked list."
      ELSE
           current  self.start
           WHILE current IS NOT NONE AND current.data < data
                current  current.next
           ENDWHILE
           IF current IS NOT NONE AND current.data = data
                THEN
                     OUTPUT "Data found!"
                     RETURN TRUE
                ELSE
                     OUTPUT "Data not found!"
                     RETURN FALSE
           ENDIF
   ENDIF
ENDFUNCTION
```

3.5 Access all nodes stored in the Linked List

```
PROCEDURE display(self)
   IF self.start IS NONE
      THEN
           OUTPUT "Empty Linked List"
      ELSE
           current  self.start
           WHILE current IS NOT NONE
                OUTPUT current.data
                current  current.next
           ENDWHILE
   ENDIF
ENDPROCEDURE
```

4 Linked list using array implementation

One reason why linked list is preferable to array is that only pointers need to be changed in a linked list when reordering is required, whereas all data items would need to be shifted for insertions or deletions at interior positions of an array.

With the extra storage space for pointers, we can implement linked lists using arrays without the expensive shifting of elements within the arrays.

Linked list can be implemented by having an array of nodes. Each node contains data and a pointer. The pointers then link up the nodes by storing the index of the next node in the array.

For the follow examples, we will set the null pointer to be -1.

An empty linked list is initialised by setting the `start` pointer to -1.

Diagram shows an empty linked list

New nodes added into the linked list are stored in contiguous locations, however, the pointers link up the nodes in an ordered sequence. The `start` pointer points to the index of the first node, which is at index 1 in the diagram below. The last node has its pointer pointing to -1.

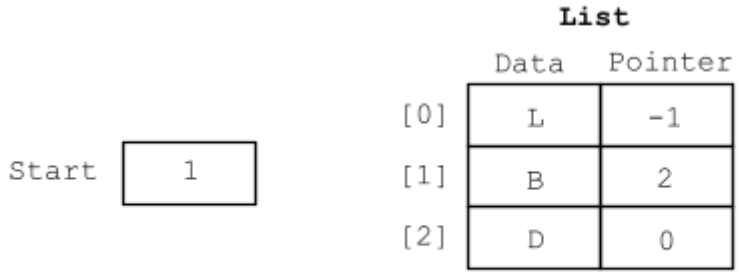
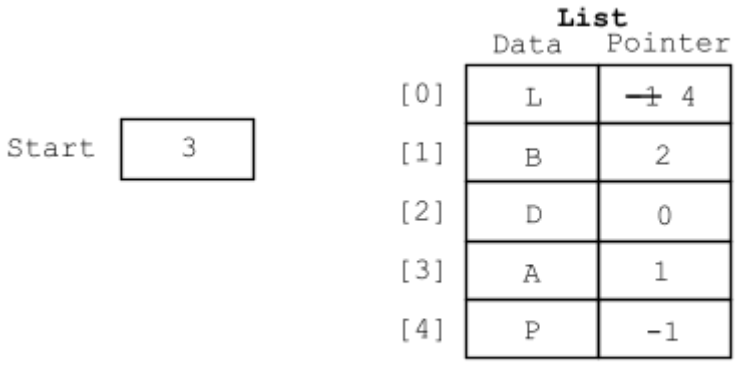


Diagram shows an ordered linked list in an array

4.2 Inserting a node in an ordered list

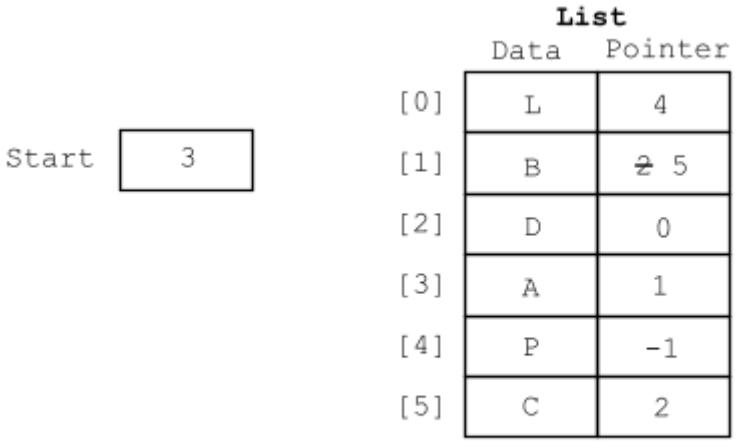
To insert a new node, A, at the beginning of the linked list:
Step 1: Insert node A into array at the next available location. In this example, at index 3.
Step 2: Set pointer of A to point to the index of the `start` pointer
Step 3: Adjust the `start` pointer to point to the index of new node A

To insert a new node, P, at the end of a linked list:
Step 1: Insert node P into array at the next available location. In this example, at index 4.
Step 2: Adjust the pointer of the last node to point to the new node at index 4.



Adding a new node to the end of a linked list

To insert a node, C, into the list:
Step 1: Insert node C into the array at the next available location at index 5.
Step 2: Set the pointer of C to point to the index that the pointer of node B is pointing at,
Step 3: Set the pointer of node B to point to the index of node C.

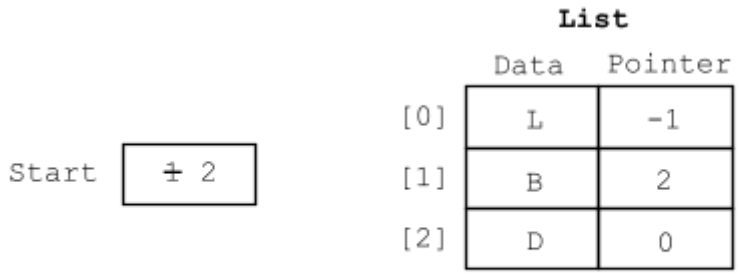


Adding a new node in between 2 nodes

4.3 Deleting a node

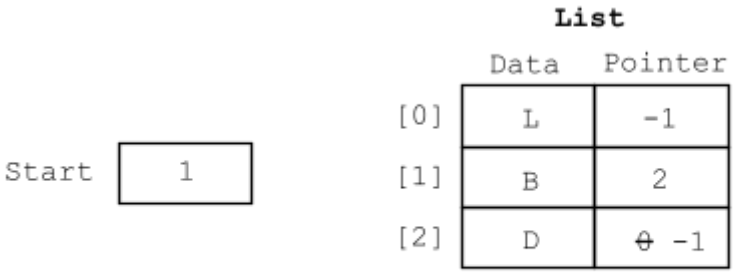
When deleting a node, only pointers need to be adjusted. The old data can remain in the array, but it will no longer be accessible as no pointer will point to it.

The diagram below shows how the `start` pointer is adjusted to delete the first node of the linked list. The `start` pointer now contains the value of the deleted node.



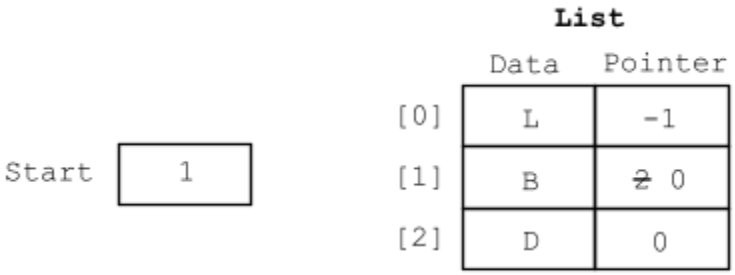
Deleting the first node of the linked list

To delete the last node in the linked list, the pointer of the node before the last node is set to -1.



Deleting the last node of the linked list

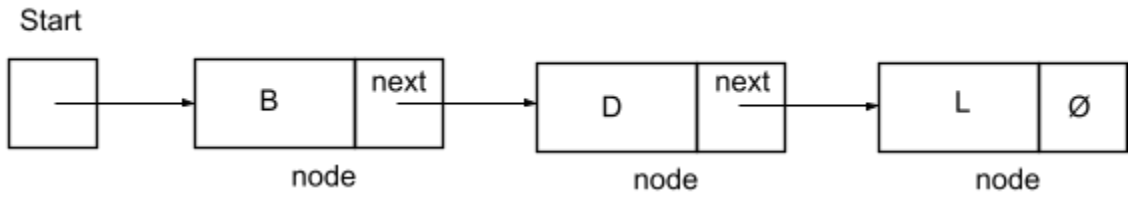
To delete a node D that is between node B and node L:
Step1: Set the pointer of B to point to the index that the pointer in D is pointing to.



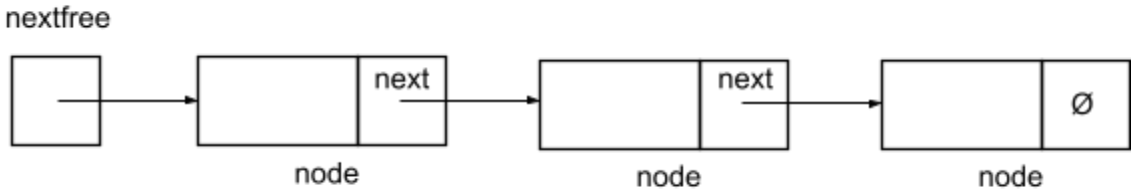
Deleting a node within the linked list

5 Free space list

As we can see from section 4.2, Nodes that are deleted will remain in the array and no longer be accessible as no pointer will point to it. This space, which is free, will then be wasted. One way of managing the free space is to keep two linked lists: one for the actual data, and one for the free space. A free space list can be created by linking unused nodes (empty nodes initially). In this manner, new nodes can be created using the unused nodes and nodes that are deleted will be returned to the free space list and can be reused again.



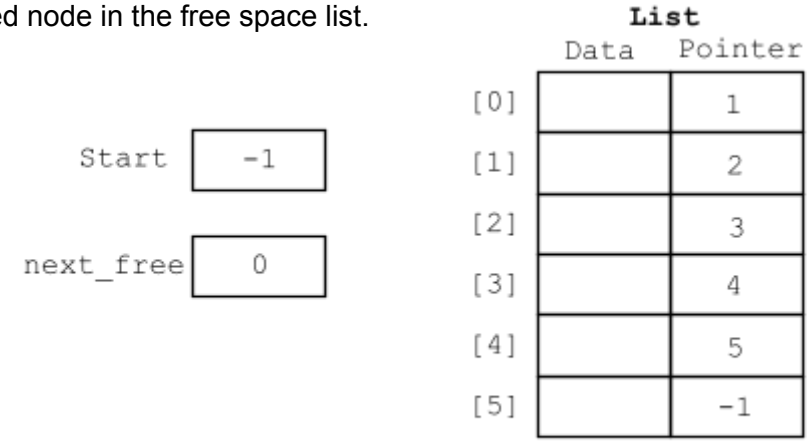
Conceptual diagram of a linked list with data with a `start` pointer



Conceptual diagram of a free space list with a `next_free` pointer

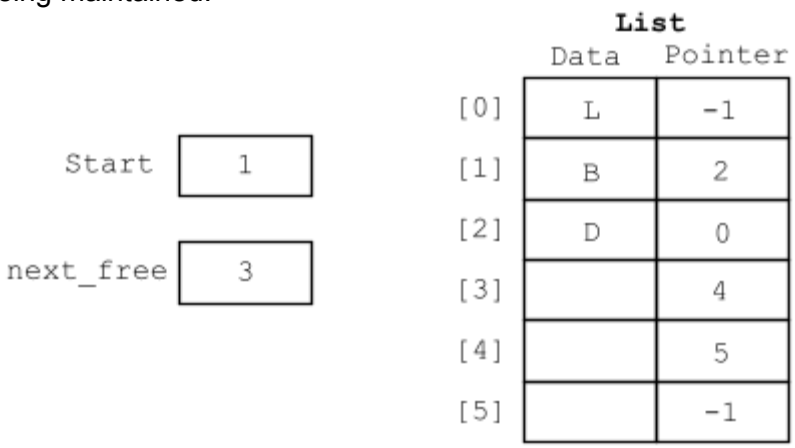
For the follow examples, we will implement the free space list using arrays. We will need to create an array of empty nodes. Data in empty nodes are set to null or empty strings. The empty nodes are then linked up using the pointers.

When an array of empty nodes is first initialised to work as a linked list, the linked list is considered to be empty, with the `start` pointer set as the null pointer, -1. All unused nodes need to be linked to form the free space list. The `next_free` pointer will point to the first unused node in the free space list.

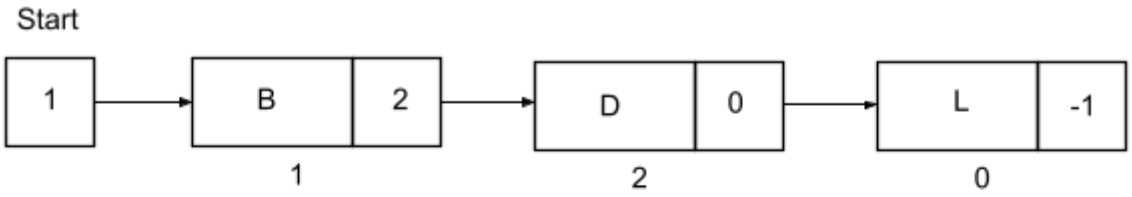


An empty linked list and a free space list of 6 linked unused nodes

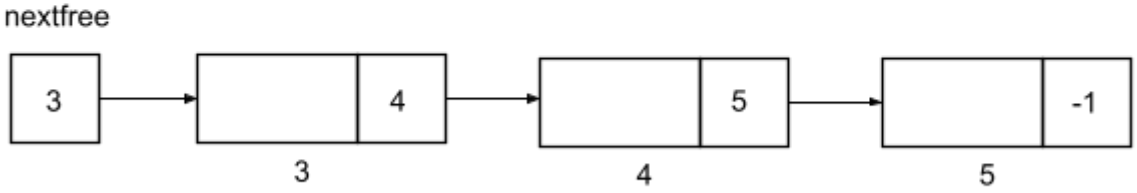
Assume data “L”, “B” and “D” were added to the linked list and be kept in alphabetical order. The diagram below shows how a linked list with data and a free space list of unused nodes are being maintained.



Linked list with data and free space list



Conceptual diagram of a linked list with data with a `start` pointer

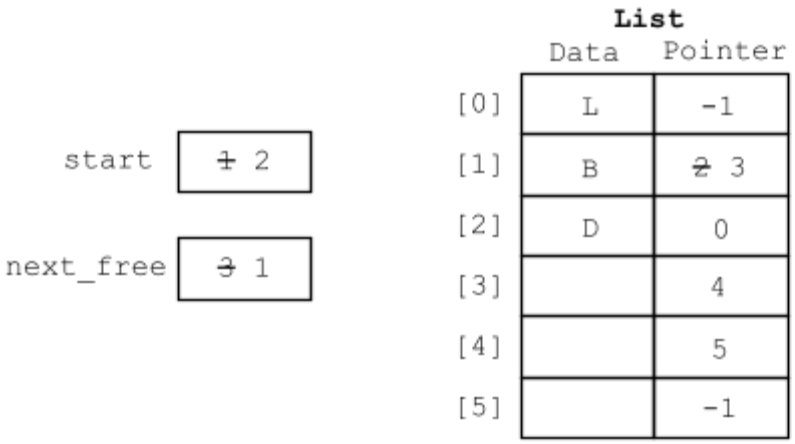


Conceptual diagram of a free space list with a `next_free` pointer

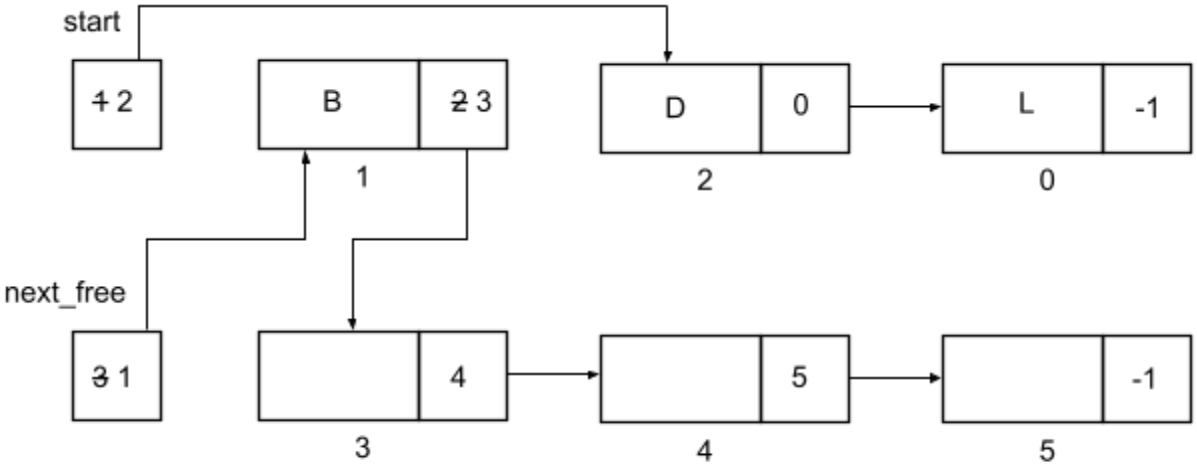
To insert data, we need to:

1. Check if there is any unused nodes in the free space list
2. Set a `new_node_pointer` to point to the index that the `next_free` pointer is pointing to.
3. Store the required data into the new node at the index that `new_node_pointer` is pointing to.
4. Set `next_free` pointer to point to index of next free node
5. Traverse the linked list find the correct position to insert new node.
6. To insert new node to the start of linked list, set the `next` pointer of new node to point to the index that `start` pointer is pointing to and set `start` pointer to point to index of new node.
7. To insert new node within the linked list, set the `next` pointer of new node to point to the index of the next node and set the pointer of the previous node to point to the index of new node.

If node B is to be deleted, the array element of that node needs to be linked back into the free space list. This space, at index 1, will now be accessible and can be reused again.



Linked list and free space list implemented using arrays



Conceptual diagram of deleting a node from linked list and adding the deleted node back to free space list as an unused node.

To delete node, we need to

1. Check if listed list is empty
2. Traverse the linked list find if data exists in the nodes
3. To delete the first node, set the `start` pointer to point to the index of the `next` pointer in the first node. Set the index of the `next` pointer in the first node to point to the index of the next free node and set the `next_free` pointer to point to the index of the node to be deleted.
4. To delete a node within the linked list, set the pointer of the previous node to point to the index of the next pointer in the node to be deleted. Set the `next` pointer in the node to be deleted to point to the index of next free node and set the `next_free` pointer to point at index of deleted node.

6 Memory allocation

Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution. There are two types of memory allocations:

- Static memory allocation
- Dynamic memory allocation

Static Memory Allocation	Dynamic Memory Allocation
Memory is allocated at compile time.	Memory is allocated at run time.
In static memory allocation, the size and location of memory blocks are fixed and cannot be changed at runtime.	In dynamic memory allocation, the size and location of memory blocks can be changed according to the program logic and data size.
Allocated memory remains from start to end of the program.	Allocated memory can be released at any time during the program.
It is inflexible and wasteful if actual memory usage is less than allocated.	It is flexibility and efficiency as memory is allocated according to required usage
It is fast and simple, as there is no need to allocate or deallocate memory during execution, and it avoids memory fragmentation, as the memory blocks are contiguous and aligned	It is slower and more complex, as you have to manage the memory allocation and deallocation yourself. It may also cause memory fragmentation.
Static memory allocation is preferred in an array.	Dynamic memory allocation is preferred in the linked list.