Temasek Junior College

2022 JC1 H2 Computing

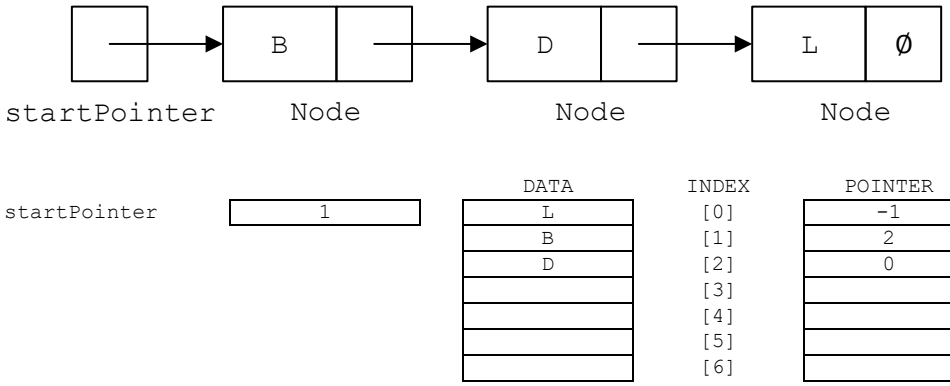
Data Structures 5 – Linked Lists (II)

§5.1 Implementing Linked Lists using 1-D Arrays

A linked list can be implemented using two 1-D arrays, one to store the data of each node in the linked list and one to store its pointer that will point to the address of the location in memory of the next node.

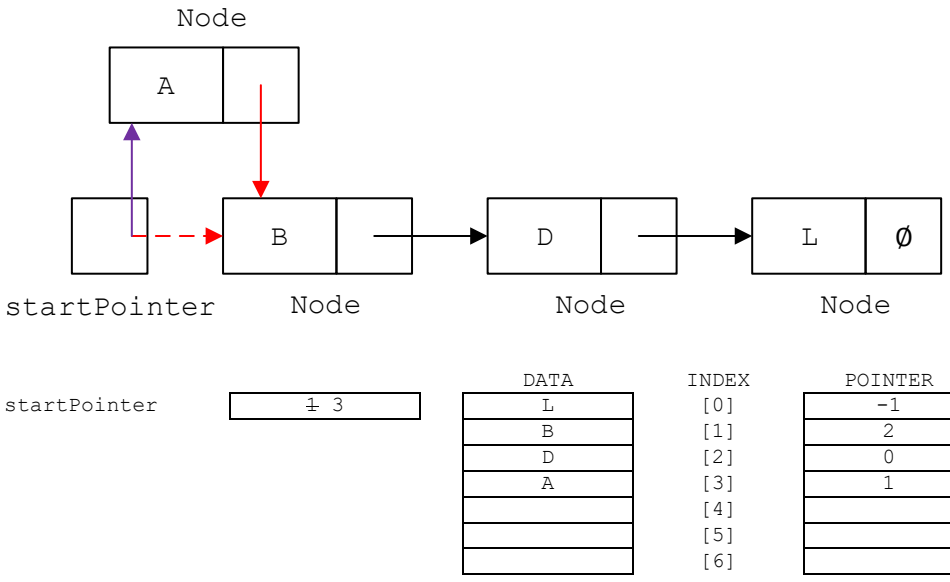
As an array has a finite size, the linked list may become full eventually and this must be allowed for.

The illustrations below shows how two 1-D arrays can be used to implement the linked list below. An element in the `DATA` array and an element in the `POINTER` array with the same `INDEX` forms a node. The value `-1` is used to represent the null pointer.



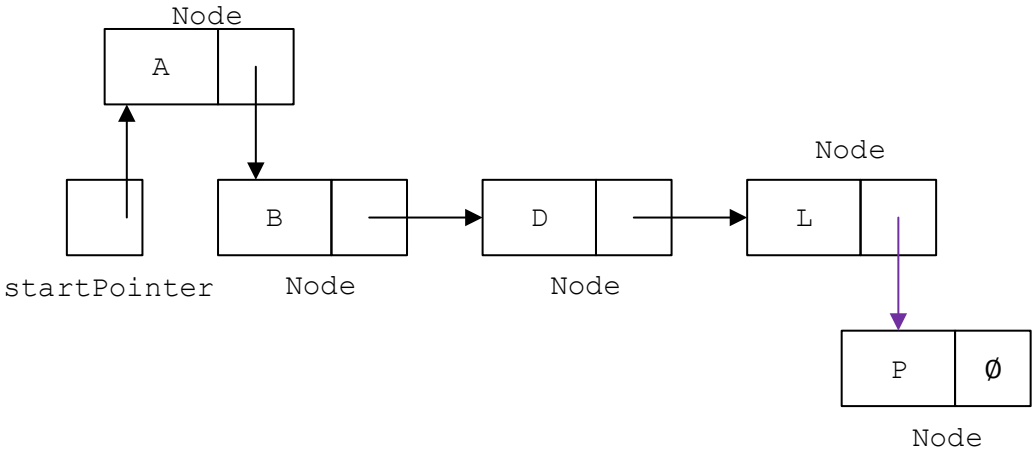
Adding a node at the beginning of the linked list

- A new node `A` is being added to the linked list at `INDEX [3]`, where the `POINTER` is set to `1` to point to the address of the location in memory of node `B`.
- `startPointer` is adjusted to value `3` to point to the address of the location in memory of node `A`.



Adding a node at the end of the linked list

- A new node `P` is being added to the linked list at `INDEX [4]`.
- Node `L`, which originally contains the null pointer, will have the pointer value adjusted to `4` to point to the address of the location in memory of node `P`.

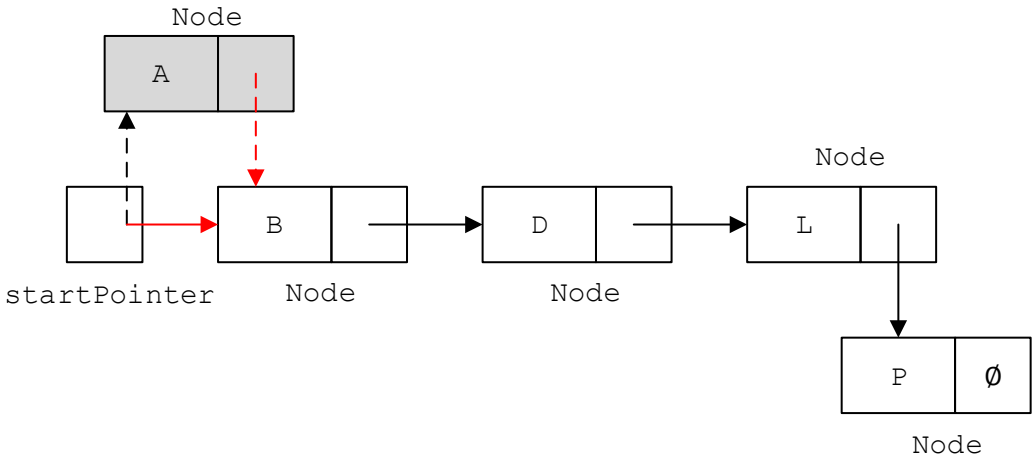


startPointer	3

DATA	INDEX	POINTER
L	[0]	~~-1~~ 4
B	[1]	2
D	[2]	0
A	[3]	1
P	[4]	-1
	[5]	
	[6]	

Deleting a node at the start of the linked list

- Node `A` is deleted.
- `startPointer` is adjusted to value `1` so as to point to the address of the location in memory of node `B`.
- The data in node `A` remains but it will no longer be accessible since there will be no pointers pointing to the address of its location in memory.

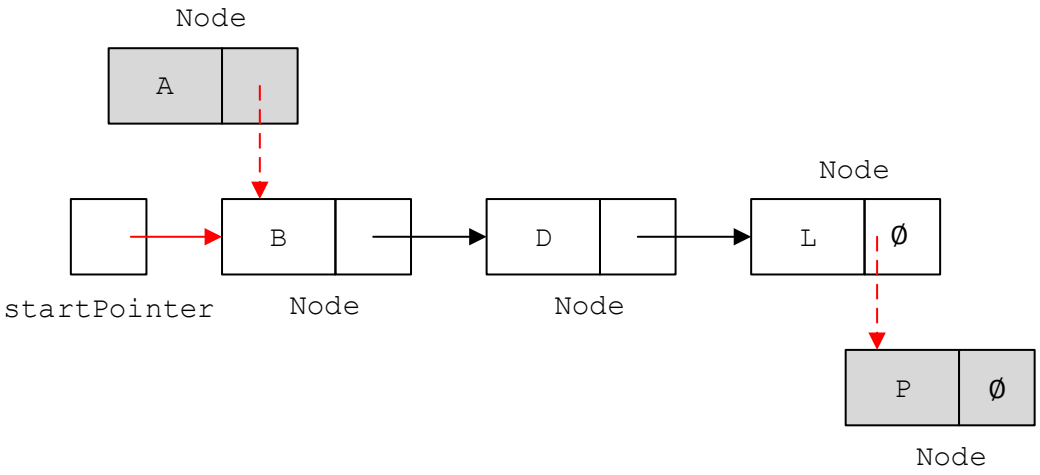


startPointer	~~3~~ 1

DATA	INDEX	POINTER
L	[0]	4
B	[1]	2
D	[2]	0
A	[3]	1
P	[4]	-1
	[5]	
	[6]	

Deleting a node at the end of the linked list

- Node `P` is deleted.
- Node `L`, which originally contains the pointer pointing to the address of location in memory of node `P` , will have its pointer value adjusted to `-1`, as it is now the end of the linked-list.
- The data in node `P` remains but it will no longer be accessible since there will be no pointers pointing to the address of its location in memory.

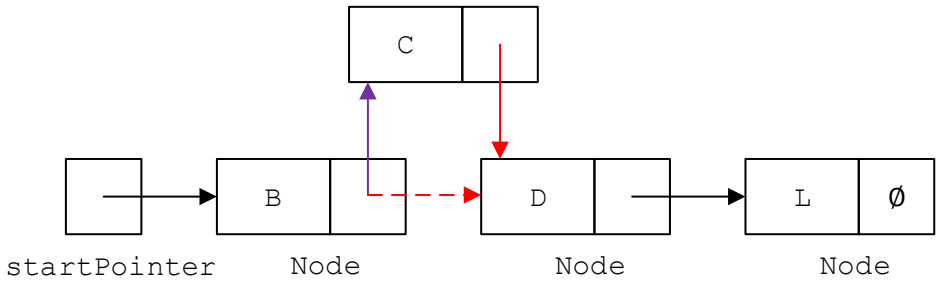


startPointer	1

	DATA	INDEX	POINTER
	L	[0]	~~4~~ -1
	B	[1]	2
	D	[2]	0
	A	[3]	1
	P	[4]	-1
		[5]	
		[6]	

Adding a node between two nodes of the linked list

- Add a new node `C`, in between nodes `B` and `D`.
- Node `B`, which originally contains the pointer pointing to the address of location in memory of node `D`, will have its pointer value adjusted to `5`, so as to point to the address of the location in memory of node `C`.
- Pointer in node C will point to the address of the location in memory of node `D`.

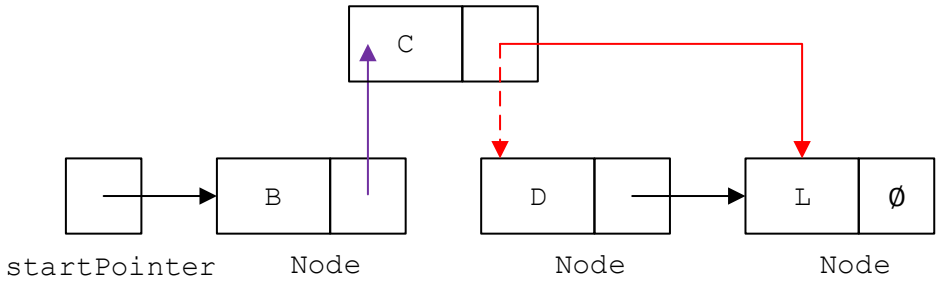


startPointer	1

	DATA	INDEX	POINTER
	L	[0]	-1
	B	[1]	~~2~~ 5
	D	[2]	0
	A	[3]	1
	P	[4]	-1
	C	[5]	2
		[6]	

Deleting a node between two nodes of the linked list

- Delete node `D`.
- Node `C`, which originally contains the pointer pointing to the address of location in memory of node `D` , will have its pointer value adjusted to `0`, so as to point to the address of the location in memory of node `L`.
- The data in node `D` remains but it will no longer be accessible since there will be no pointers pointing to the address of its location in memory.



startPointer	1

	DATA	INDEX	POINTER
	L	[0]	-1
	B	[1]	5
	D	[2]	0
	A	[3]	1
	P	[4]	-1
	C	[5]	~~2~~ 0
		[6]	

§5.2 Free-Space List

To ensure efficiency during run-time, the unused nodes in a linked-list implemented using arrays must to be easy to locate. To do so, these unused nodes can be linked up to form another linked list, which becomes the **free space list** (or **heap**).

When an array of nodes is first initialised to work as a linked list, the linked list will be empty. The `startPointer` will be the null pointer initially while the `freeListPointer` will pointer to the first node.

Since the linked list is empty initially, all the nodes will be linked up to form the free space list.

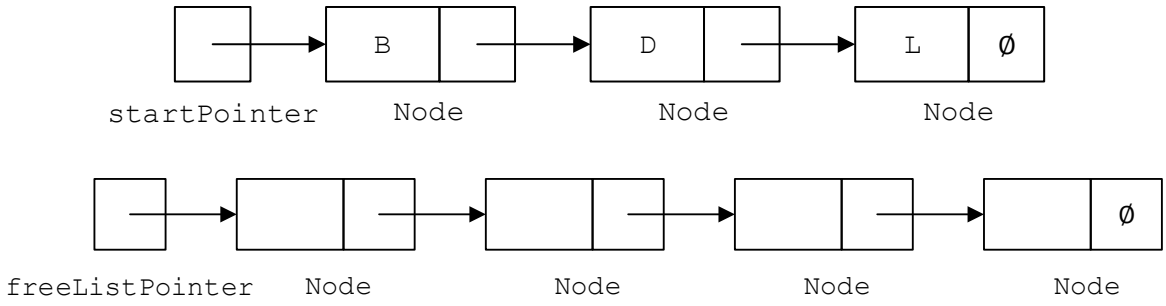
startPointer	-1
freeListPointer	0

	DATA	INDEX	POINTER
		[0]	1
		[1]	2
		[2]	3
		[3]	4
		[4]	5
		[5]	6
		[6]	-1

Now, the nodes L, B and D are added to the linked list at INDEX [0], [1] and [2] respectively. Assume that the order of the nodes follows the alphabetical order regardless of the INDEX.

The startPointer will have its value adjusted to 1, which is the first node, so as to pointer to the address of the location in memory of node B.

The empty nodes now run from INDEX [3] to [6]. Hence the freeListPointer will have its value adjusted to 3, so as to point to the address of the location in memory of the first empty node.



startPointer: 1

freeListPointer: 3

	DATA	INDEX	POINTER
	L	[0]	~~0~~ -1
	B	[1]	~~1~~ 2
	D	[2]	~~3~~ 0
		[3]	4
		[4]	5
		[5]	6
		[6]	-1

When node D is deleted from the linked list, a good practice to enable the node to be used for storing future data is to link it back to the free space list at the front.

startPointer: 1

freeListPointer: ~~3~~ 2

	DATA	INDEX	POINTER
	L	[0]	-1
	B	[1]	~~2~~ 0
	D	[2]	~~0~~ 3
		[3]	4
		[4]	5
		[5]	6
		[6]	-1

The free space list facilitates a dynamic increase and decrease in size of a linked list by providing easy access to and recovery of empty nodes. Nodes can be easily added from the free space list when they are needed by the linked lists. Nodes deleted and no longer required by the linked list can also be easily recovered back to the free space list. All these are done through the adjustment of the pointer values of the nodes to allow them to reference to the correct addresses of locations in the memory. It is interesting to note that the free space list itself is also a linked list.

§5.3 Pseudocode for Setting up a Linked List

The pseudocode for setting up a linked list with 12 nodes is provided below. Note that the indexing in the pseudocode follows Python indexing.

```
DECLARE linkedList ARRAY[0:11] OF INTEGER
DECLARE linkedListPtrs ARRAY[0:11] OF INTEGER
DECLARE startPtr: INTEGER
DECLARE freeListPtr: INTEGER
DECLARE index: INTEGER

freeListPtr ← 0
startPtr ← -1 // linked list is initially empty

// when the linked list is initialised, all the spaces are empty
// this becomes the initial free space list, a linked list of all the empty spaces
FOR index ← 0 TO 11
linkedListPtrs[index] ← index + 1
NEXT index

// the final linkedListPtr is set to -1 to show no further links
linkedListPtrs[11] ← -1
```

The identifier table below gives the definitions of the identifiers used:

Identifier	Description
linkedList	Array representing the linked list
linkedListPtrs	Pointers for the linked list
startPtr	Start of the linked list
freeListPtr	Start of the free space list
index	Pointer to current element in the linked list

The value of `-1` is used to represent a null pointer.

The illustration below shows a schematic setup based on the pseudocode:

freeListPtr = 0

index	linkedList	linkedListPtrs
[0]		1
[1]		2
[2]		3
[3]		4
[4]		5
[5]		6
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

startPtr = -1

§5.4 Pseudocode for Linked List Traversal and Finding Items in a Linked List

Traversing a linked list means to work your way systematically through the linked list. You may need to do this to find an element in the list or if you want to display the contents of the list.

To traverse a linked list you need to start from the first node, follow its pointer to the next node and do so repeatedly until you reach the last node, which has a null pointer. If the start pointer is a null pointer, then the linked list is empty.

The pseudocode below gives the algorithm to find an item in the linked list. The pointer to the item is returned if it exists. If the item does not exist, the null pointer will be returned. This is because the entire linked list has been traversed from the start till the end. The null pointer belongs to that of the last node.

```
DECLARE itemSearch : INTEGER
DECLARE itemPtr : INTEGER
CONSTANT nullPtr ← -1

FUNCTION find(itemSearch) RETURNS INTEGER

// this function returns the item pointer of the value found or -1 if the item is not found

    DECLARE found: BOOLEAN
    found ← FALSE
    itemPtr ← startPtr
        WHILE (itemPtr <> nullPtr) AND NOT found
            IF linkedList[itemPtr] = itemSearch
                THEN
                    found ← TRUE
                ELSE
                    itemPtr ← linkedListPtrs[itemPtr]
            ENDIF
        ENDWHILE
    RETURN itemPtr
ENDFUNCTION
```

The identifier table below gives the definitions of the additional identifiers used:

Identifier	Description
itemSearch	Item to find
itemPtr	Variable for storing pointer of current node for traversing to next node
nullPtr	Indicates end of linked list
found	Flag to terminate the traversal if item is found

The illustration below shows a schematic setup of a populated linked list:

startPtr = 4
freeListPtr = 5

index	linkedList	linkedListPtrs
[0]	27	-1
[1]	19	0
[2]	36	1
[3]	42	2
[4]	16	3
[5]		6
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

The trace table below shows the algorithm being used to search for the item 42

startPtr	itemPtr	searchItem
4	4	42
	3	

§5.5 Pseudocode for Adding Items to a Linked List

The pseudocode below gives the algorithm for adding an item to the start of a linked list.

```
DECLARE itemAdd: INTEGER
DECLARE startPtr : INTEGER
DECLARE freeListPtr: INTEGER
DECLARE tempPtr : INTEGER
CONSTANT nullPointer ← -1

PROCEDURE linkedListAdd(itemAdd)

    // check whether linked list is full
    IF freeListPtr = nullPtr
        THEN
            OUTPUT "Linked list is full"
        ELSE
            // get next node in list from the free space list
            tempPtr ← startPtr  // keep old start pointer
            startPtr ← freeListPtr  // set start pointer to next node in free space list
            freeListPtr ← linkedListPtrs[freeListPtr]  // reset freeListPtr
            linkedList[startPtr] ← itemAdd  // put item in list
            linkedListPtrs[startPtr] ← tempPointer  // update linkedListPtrs
    ENDIF
ENDPROCEDURE
```

The identifier table below gives the definitions of the additional identifiers used:

Identifier	Description
itemAdd	Item to be added
tempPtr	Variable for storing `startPtr` before addition of item

The illustration below shows a schematic setup of a populated linked list:

startPtr = 4
freeListPtr = 5

index	linkedList	linkedListPtrs
[0]	27	-1
[1]	19	0
[2]	36	1
[3]	42	2
[4]	16	3
[5]		6
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

The trace table below shows the algorithm being used to add the item 18 to the front of the linked list:

startPtr	freeListPtr	itemAdd	tempPtr
4	5	18	
5	6		4

The linked list now becomes:

startPtr = 5
freeListPtr = 6

index	linkedList	linkedListPtrs
[0]	27	-1
[1]	19	0
[2]	36	1
[3]	42	2
[4]	16	3
[5]	18	4
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

§5.6 Pseudocode for Deleting Items from a Linked List

The pseudocode below gives the algorithm for deleting an item from a linked list.

```
DECLARE itemDelete : INTEGER
DECLARE oldIndex : INTEGER
DECLARE index : INTEGER
DECLARE startPtr : INTEGER
DECLARE freeListPtr : INTEGER
DECLARE tempPtr : INTEGER
CONSTANT nullPtr = -1

PROCEDURE linkedListDelete(itemDelete)

    // check whether linked list is empty
    IF startPtr = nullPtr
        THEN
            OUTPUT "Linked list empty"
        ELSE
            // find item to delete in linked list
            index ← startPointer
            WHILE linkedList[index] <> itemDelete AND (index <> nullPtr)
                oldIndex ← index
                index ← linkedListPtrs[index]
            ENDWHILE

            IF index = nullPtr
                THEN
                    OUTPUT "Item ", itemDelete, " not found"
                ELSE
                    // delete the pointer and the item
                    tempPtr ← linkedListPtrs[index]
                    linkedListPtrs[index] ← freeListPtr
                    freeListPtr ← index
                    linkedListPtrs[oldIndex] ← tempPtr
            ENDIF
    ENDIF
ENDPROCEDURE
```

The identifier table below gives the definitions of the additional identifiers used:

Identifier	Description
itemDelete	Item to be deleted
oldIndex	Pointer to previous node
tempPtr	Variable for temporary storing pointer of item before its deletion

The illustration below shows a schematic setup of a populated linked list before deletion:

startPtr = 5
freeListPtr = 6

index	linkedList	linkedListPtrs
[0]	27	-1
[1]	19	0
[2]	36	1
[3]	42	2
[4]	16	3
[5]	18	4
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

The trace table below shows the algorithm being used to delete 36 from the linked list:

startPtr	freeListPtr	itemDelete	index	oldIndex	tempPtr
5	6	36	5		
			4	5	
			3	4	
			2	3	
					1
	2				

The linked list now becomes:

freeListPtr = 2
startPtr = 5

index	linkedList	linkedListPtrs
[0]	27	-1
[1]	19	0
[2]	36	6
[3]	42	1
[4]	16	3
[5]	18	4
[6]		7
[7]		8
[8]		9
[9]		10
[10]		11
[11]		-1

Exercise

Implement Linked List using two 1-D arrays in Python.