Abstract Data Types (ADTs) (Part II)

Outline

- Abstract Data Types (ADTs)
  - Definition
  - ADTs vs Data Structures
  - Client vs Implementation
  - Benefits
- Implementations of ADTs
  - Stack
    - Implementation using array and linked list
    - Application: Postfix Notation
  - Queue
    - Array and Linked List
    - Application: Breadth-first search on a graph
    - Priority queue

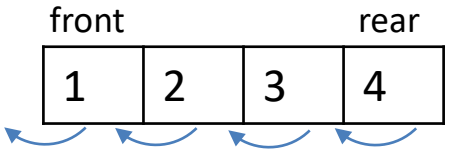
Queue

- A queue is an ADT that adopts a **FIFO** (first-in first-out) policy.
- Two basic operations:
  - Add (push) a new item to the rear
  - Remove (pop) the item at the front (one that has been in the queue the longest)

Implementation: Array or Linked List

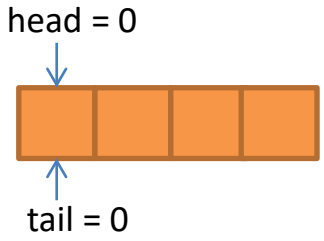
- Linked List
  - Constant time for removing from the front: $O(1)$
  - Expensive to add to the rear using a just a head pointer: $O(n)$
  - Use a tail pointer for adding to the rear: $O(1)$
  - Implementing a FIFO Queue using a linked list is trivial.

- Array
  - Constant time for adding to the end: $O(1)$
  - Expensive to remove from the front: $O(n)$
  - Use a circular array.
  - Implementation using an array (efficiently) is slightly more interesting.

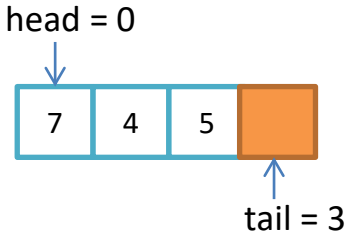


Queue Implementation using a Circular Array

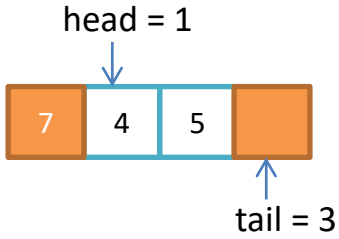
- We will use
  - `head`: index of the first item (front) of the queue
  - `tail`: index after the position of the last item (rear) of the queue
- Initially: head = tail = 0, indicating an empty queue.
- When we add an item, we increment tail.
- When we remove an item, we increment head.
- `head` and `tail` will "wrap around" at the end of the queue. This leads to a circular array.
- A circular array gives us $O(1)$ for both adding and removing.



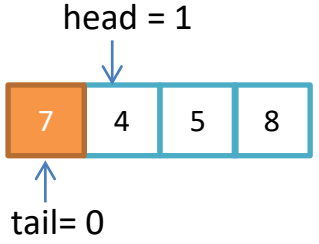
Initially: head = tail = 0



Add: increment tail



Remove: increment head.

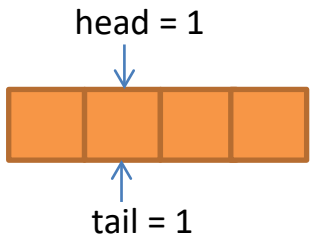


"Wrap around" at the end of the queue

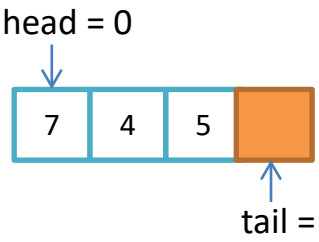
Implementation using a Circular Array

- We will use an array of `SIZE` elements.
- We keep one unused slot to distinguish between full and empty.
  - Array of size N can only hold N-1 elements.

- We have to keep track of the start and end of the array.
  - `if (tail == head), the queue is empty.`
  - `if ((tail+1)%SIZE==head), the queue is full.`
  - Number of items in queue is `(tail-head+SIZE)%SIZE`.

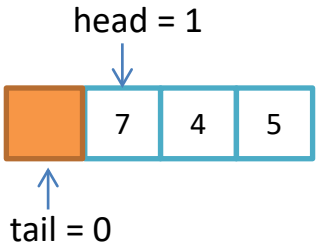


Empty queue
`head = tail`



Full queue: `(3+1)%4==0`

No. items: `(3-0+4)%4=3`

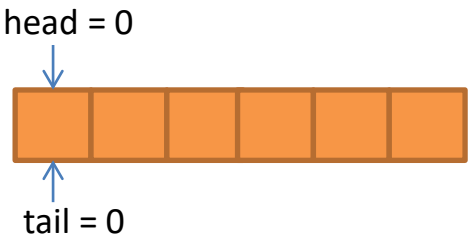


Full queue: `(0+1)%4==1`

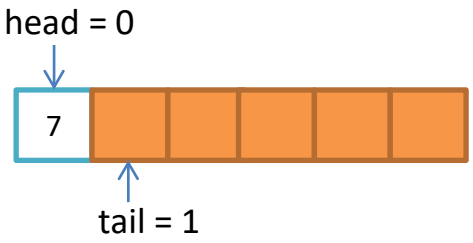
No. items: `(0-1+4)%4=3`

Queue Operations Example

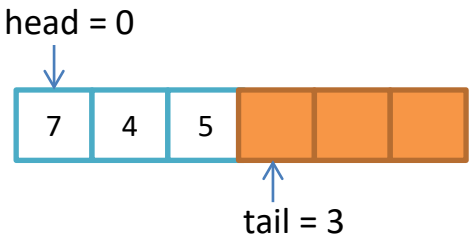
```
Queue queue(5);
```



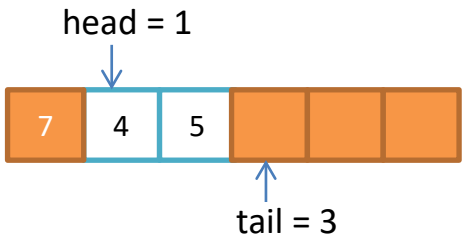

```
queue.Push(7);
```



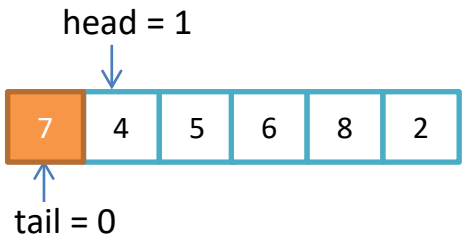

```
queue.Push(4);
queue.Push(5);
```



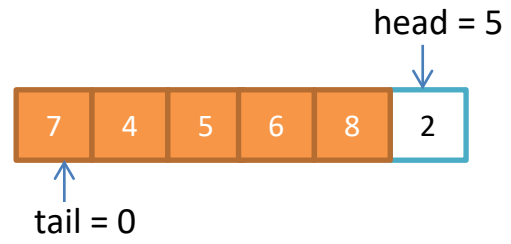

```
queue.Pop(); // 7
```




```
queue.Push(6);
queue.Push(8);
queue.Push(2);
```

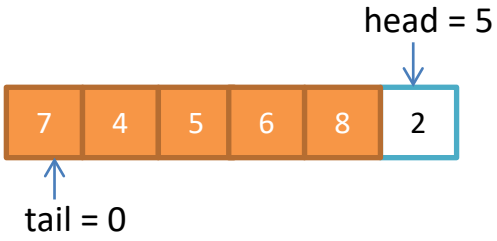



```
queue.Pop(); // 4
queue.Pop(); // 5
queue.Pop(); // 6
queue.Pop(); // 8
```

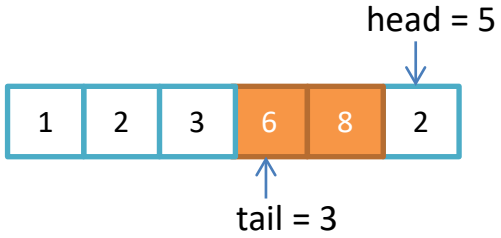


QUEUE IS FULL

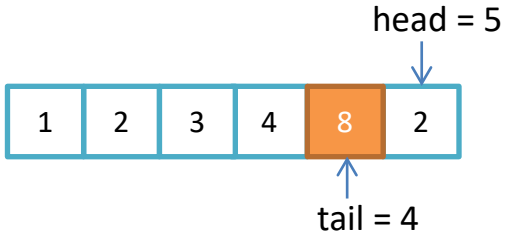
Queue Operations Example



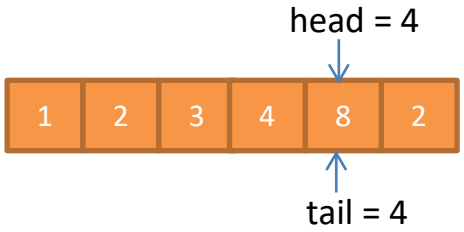

```
queue.Push(1);
queue.Push(2);
queue.Push(3);
```




```
queue.Push(4);
```




```
Queue.Remove(); // 2
Queue.Remove(); // 1
Queue.Remove(); // 2
Queue.Remove(); // 3
Queue.Remove(); // 4
Queue.Remove(); // empty!
```



Queue Implementation using a Circular Array

```
template <typename T>

class Queue{
   private:
      T *items;
      int head;
      int tail;
      int SIZE;

   public:
      Queue (int maxItem){
         SIZE = maxItem + 1;
         items = new T [SIZE];
         head = 0;
         tail = 0;

      }

    bool IsEmpty(void) const {
        return (tail == head);
     }

    bool IsFull(void) const{
        return ((tail+1)%SIZE==head);
    }
```

```
void Push(T item){
      if (!IsFull()){
         items[tail++] = item;
         if (tail==SIZE)
         {
            tail = 0;
         }
      }
   }

   T Pop(void){
      if (!IsEmpty()) {
         T temp = items[head];
         head++;
         if (head==SIZE){
            head = 0;
         }
         return temp;
      }
   }
};
```

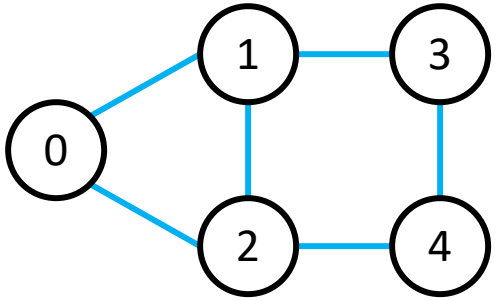
Choice of Queue Implementation

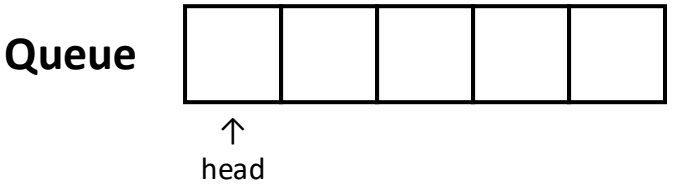
- Circular array: fixed size, costly resizing, better cache locality, lower memory overhead
- Linked list: dynamic size, flexibility, more memory overhead per element

- The choice between a circular array-based queue and a linked list-based queue depends on the specific requirements of the application. E.g.,
  - For application where cache locality is important (e.g., real time system), a circular array based queue might be more suitable.
  - If the size of the queue is not known in advance and may vary dynamically, a linked list-based queue can be more flexible.

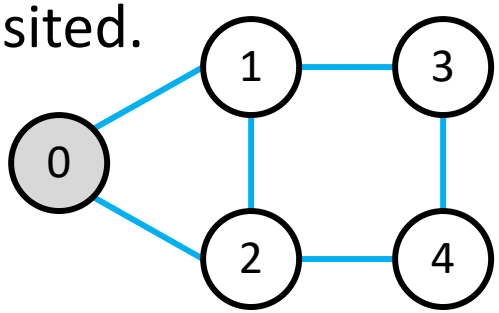
Queue Application: Breadth-first Search (BFS) on a Graph

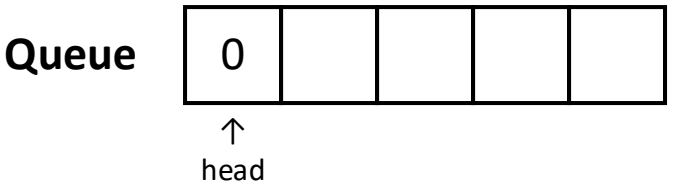
- BFS is used for traversing or searching a graph.
- All vertices at the current level are visited before moving on to vertices at the next level.
- Step 1: Initial queue is empty.





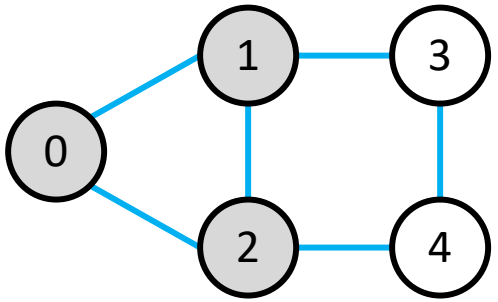
- Step 2: Start from node 0 and push it into the queue and mark it visited.

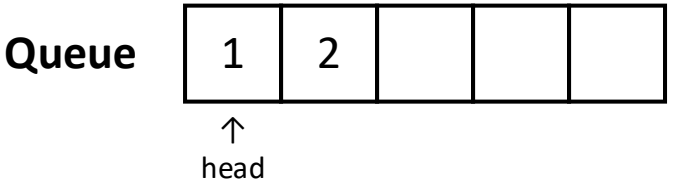




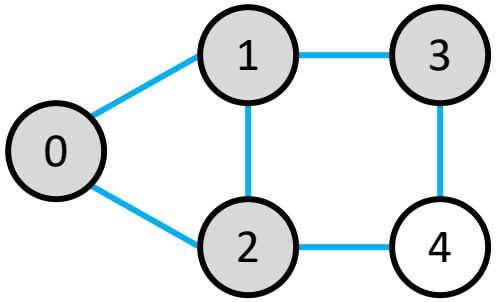
Queue Application: Breadth-first Search on a Graph

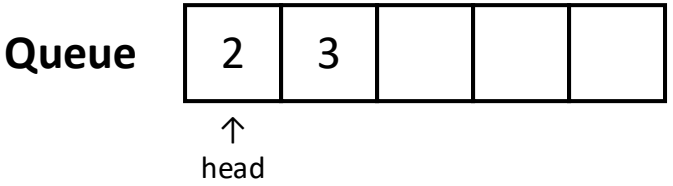
- Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.





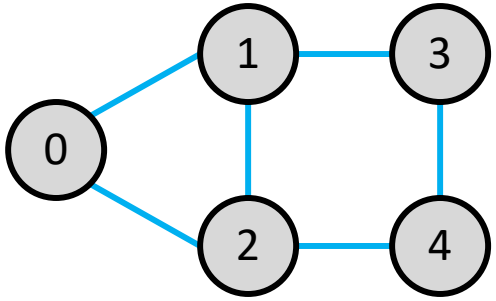
- Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

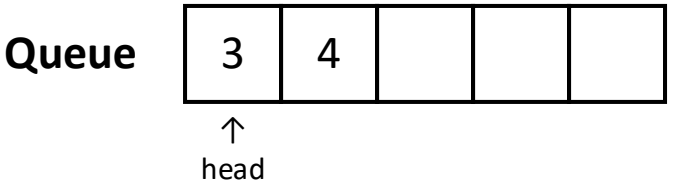




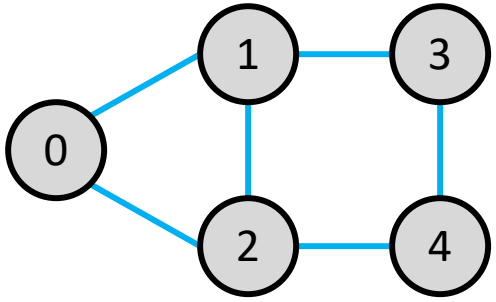
Queue Application: Breadth-first Search on a Graph

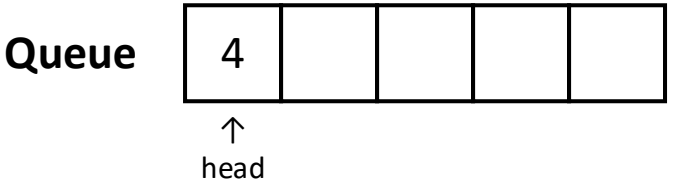
- Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.





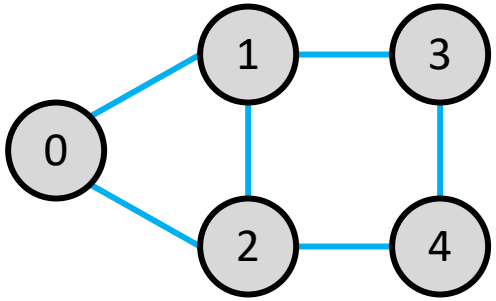
- Step 6: remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

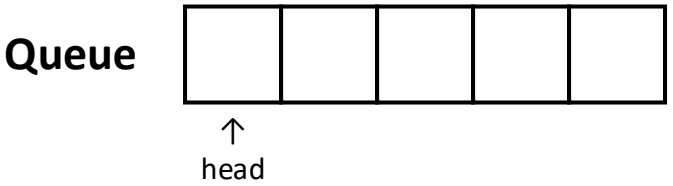




Queue Application: Breadth-first Search on a Graph

- Step 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.





Priority Queue

- Priority queue is an ADT that pushes a new item to the rear and pops the item with the highest priority.
  - Priority is user-defined
    - Could be the item with the highest id.
    - Could be the item with the oldest timestamp (think of messaging systems).
- It can be implemented using an array or a linked-list.

Priority Queue: Public Interface

```
class PriorityQueue{
   private:
   // private data

   public:
      PriorityQueue(int capacity);
      ~PriorityQueue(void);
      void Push(int Item);
      int Pop(void);
      bool IsEmpty(void) const;
      bool IsFull(void) const;
      void Dump(void) const;
};
```

Priority Queue: Private Interface

```
class PQArray{
   private:
      int *array_;
      int capacity_;
      int count_;
   public:
   // public interface
};
```

```
struct PQNode{
   PQNode *next;
   int data;
};


class PQList{
   private:
      PQNode *list_;
      int capacity_;
      int count_;
   public:
   // public interface
};
```

Priority Queue

- Complexity depends on whether the list/array is sorted or unsorted.

Operations	Unsorted	Sorted
Push	$O(1)$	$O(n)$
Pop	$O(n)$	$O(1)$

Priority Queue using a Sorted Linked List

```
int main(void){
    PQList pq(10);
    // Sorted linked list implementation

    pq.Add(4);  pq.Add(7);  pq.Add(2);
    pq.Add(5);  pq.Add(8);  pq.Add(1);

    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    return 0;
}
```

Output:

8 7 5 4 2 1

Removing: 8

7 5 4 2 1

Removing: 7

5 4 2 1

Removing: 5

4 2 1

Priority Queue using an Unsorted Array

```
int main(void){
    PQArray pq(10);
    // Unsorted array implementation

    pq.Add(4);  pq.Add(7);  pq.Add(2);
    pq.Add(5);  pq.Add(8);  pq.Add(1);

    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    return 0;
}
```

Output:

4 7 2 5 8 1

Removing: 8

4 7 2 5 1

Removing: 7

4 1 2 5

Removing: 5

4 1 2

Priority Queue

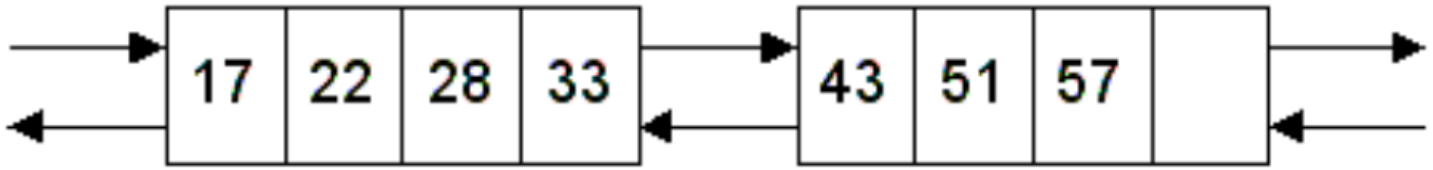
- Result is the same.
- Implementation are different.
- Complexities are different.

Summary

- Abstract Data Types (ADTs)
  - Definition
  - ADTs vs Data Structures
  - Client vs Implementation
  - Benefits
- Implementations of ADTs
  - Stack
    - Implementation using array and linked list
    - Application: Postfix Notation
  - Queue
    - Array and Linked List
    - Application: Breadth-first search on a graph
    - Priority queue

BList

- Also known as unrolled linked list
- A linear data structure
- A linked list of arrays of the same size
- Example: A BList of 4 items per node



Advantages of BList

- Linked list
  - Advantages
    - Insertion and deletion: $O(1)$
  - Disadvantages
    - Traversal is slow due to poor cache locality: $O(n)$
    - Memory overhead for storing references
- Advantages of BList over linked list
  - Faster traversal due to better cache locality
  - Reduced memory overhead to store references

BList Operations

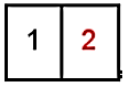
- Push back
- Push front
- Insert
- Split

BList Operations: Push back

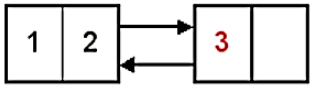
- Add an element to the back of a Blist.

Push back 1:

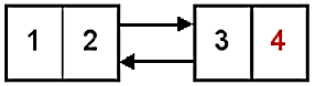
Push back 2:



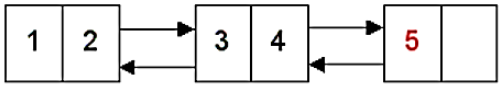
Push back 3:



Push back 4:



Push back 5:



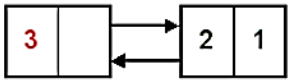
BList Operations: Push front

- Add an element to the front of a BList

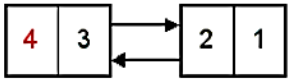
Push front **1**:

Push front **2**:

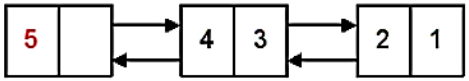
Push front **3**:



Push front **4**:



Push front **5**:

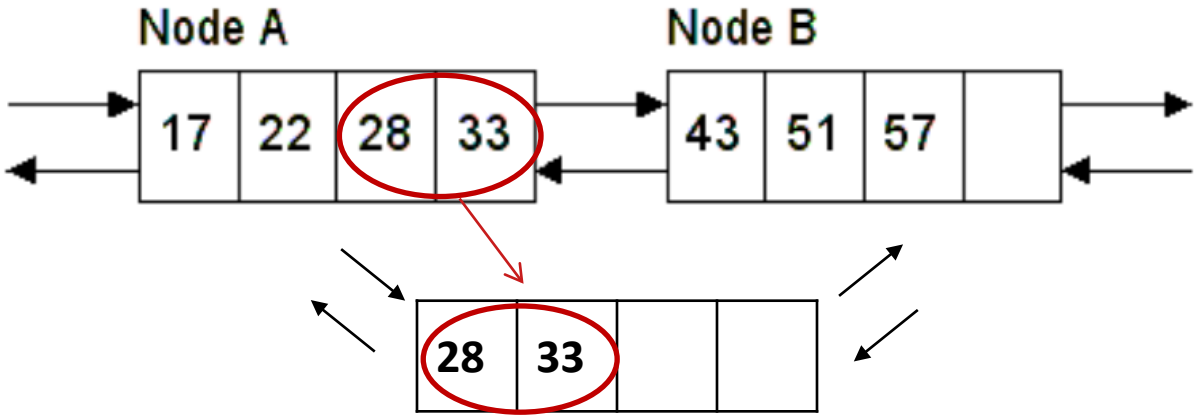


BList Operations: Insertion

- Insert an element to a sorted Blist.
- First, find the node where the element is to be inserted.
  - The node to be inserted into is not full.
  - The node to be inserted into is full.

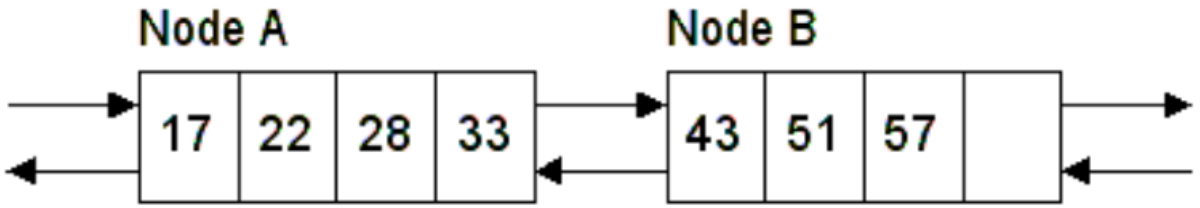
BList Operations: Split

- If the node to be inserted ino is a full node, split it into two.
  - Insert a new node after the full node.
  - Move 2nd half of the elements in the full node to the 1st half of the new node

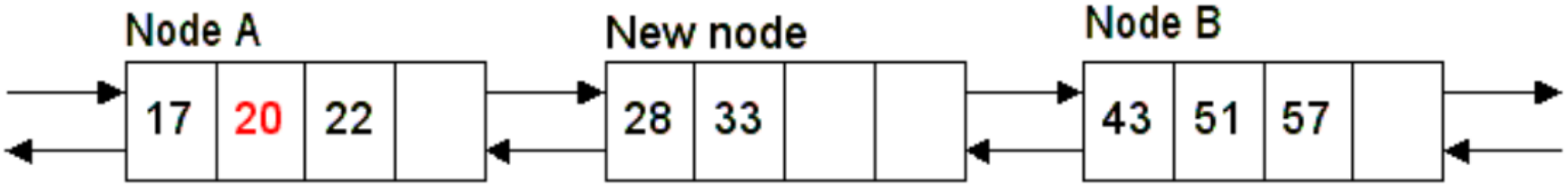


BList Operations: Insertion

- Insert 20
  - Before insertion

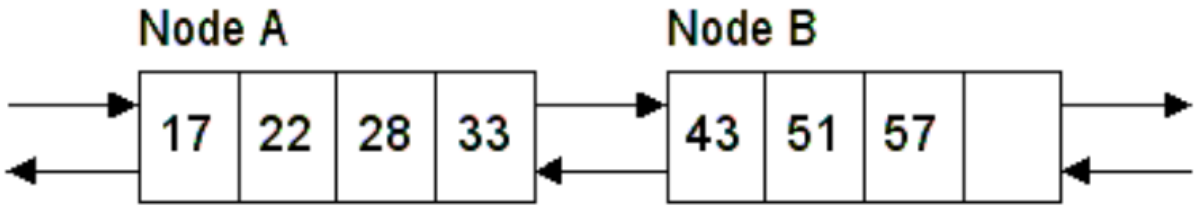


  - After insertion

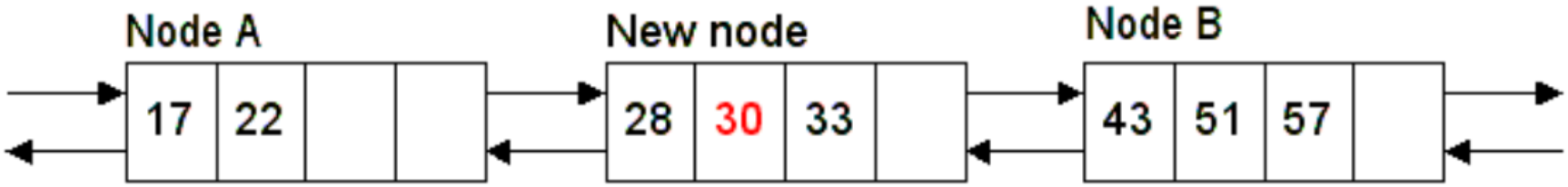


BList Operations: Insertion

- Insert 30
  - Before insertion

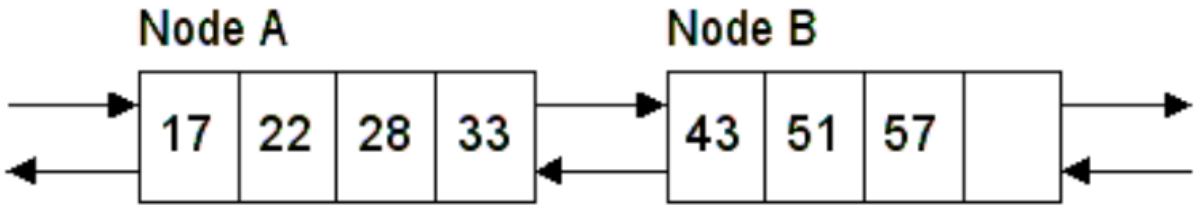


  - After insertion

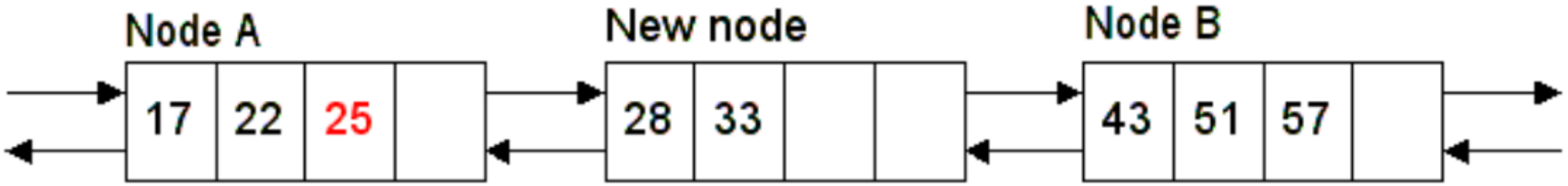


BList Operations: Insertion

- Insert 25
  - Before insertion



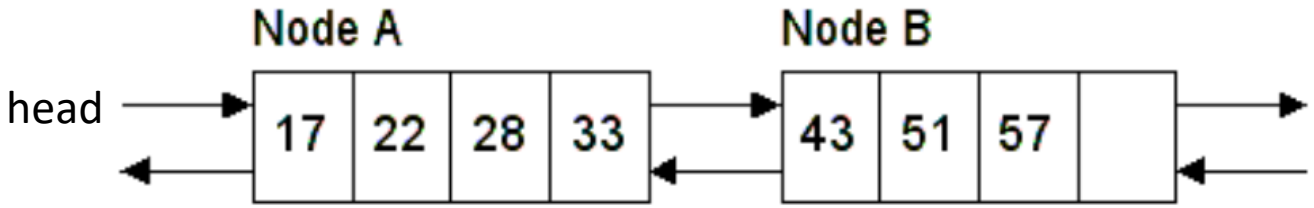
  - After insertion



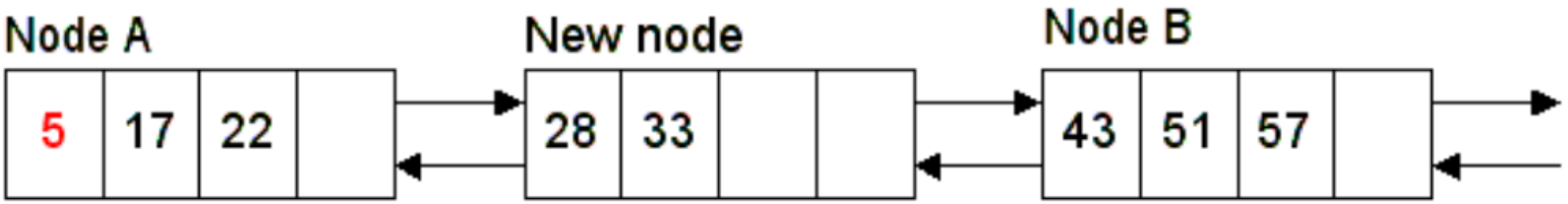
Even though the value 25 can go in either node above (both have room), we want to minimize the shifting of existing elements.

BList Operations: Insertion

- Insert 5
  - Before insertion



  - After insertion

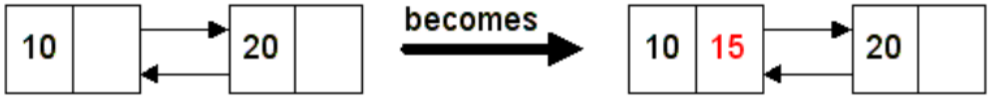


BList Operations: Insertion

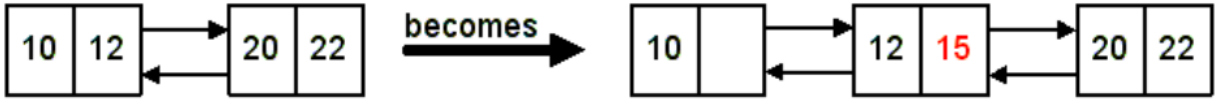
- Which node to insert?

Inserting the value 15 in four different cases.

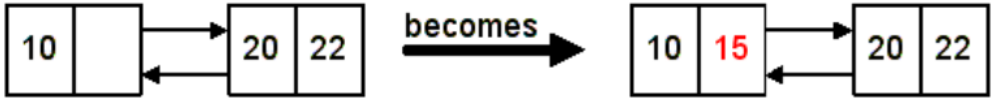
1. Both the left and right nodes have room:



2. Both the left and right nodes are full: (Split the left node)



3. The left node has room and right node is full:



4. The left node is full and the right node has room: