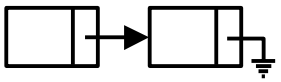
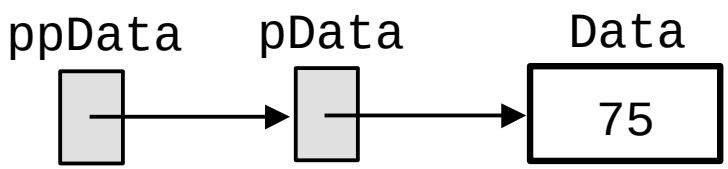
(Part II)

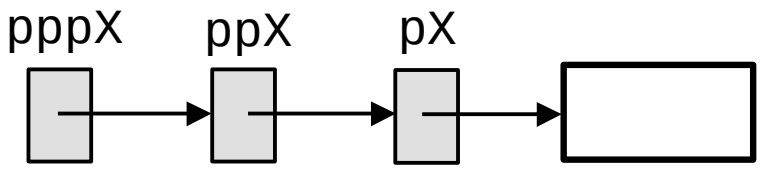
# LINKED LISTS



# Call by Reference



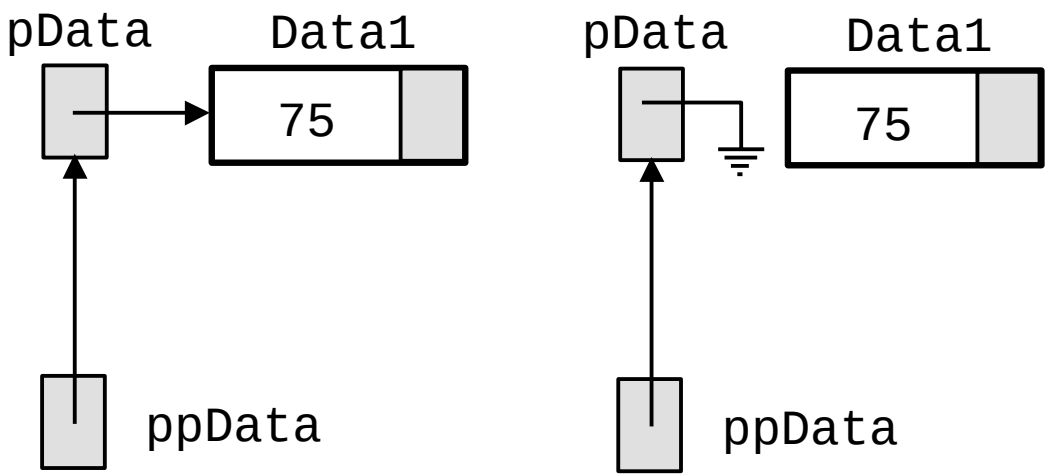

```
int    Data, *pData;
int    **ppData;

Data   = 75;
pData  = &Data;
ppData = &pData;

Data++;
(*pData)++;  (**ppData)++;

Inc( &Data );
IncP( &pData );
IncPP( &ppData );
```




```
void Inc( int *pX )
{
  return( (*pX)++ );
}

void IncP( int **ppX )
{
  return( (**pX)++ );
}

void IncPP( int ***pppX );
{
  return( (***pppX)++ );
}
```

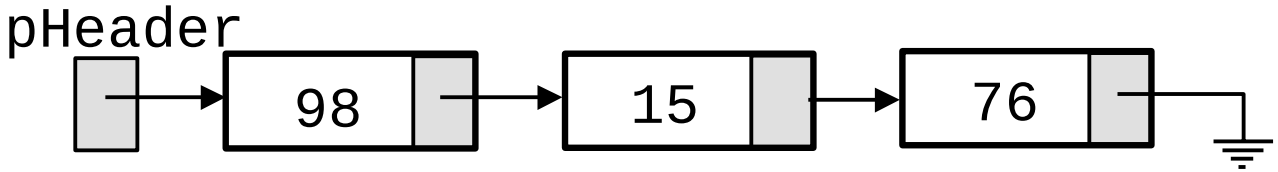
# Call by Reference

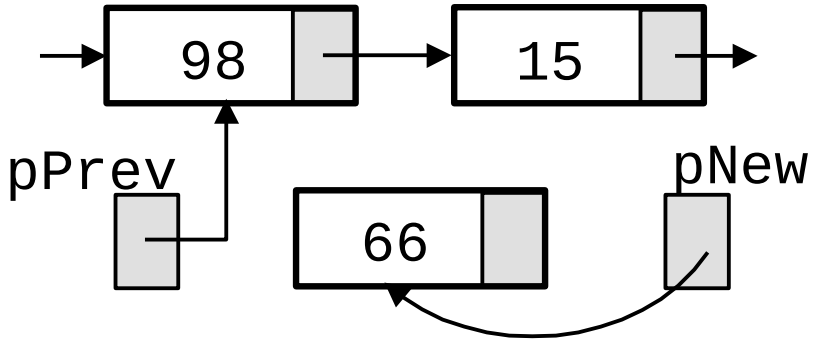
```
NodeType  Data1;
NodeType  *pData;

Data1.info = 75;
pData  = &Data;

(Data1.info)++;
(pData1->info)++;

MakeNULL( &pData );
```

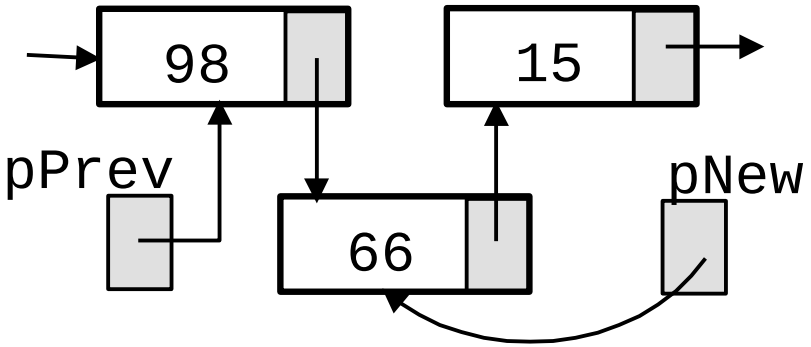


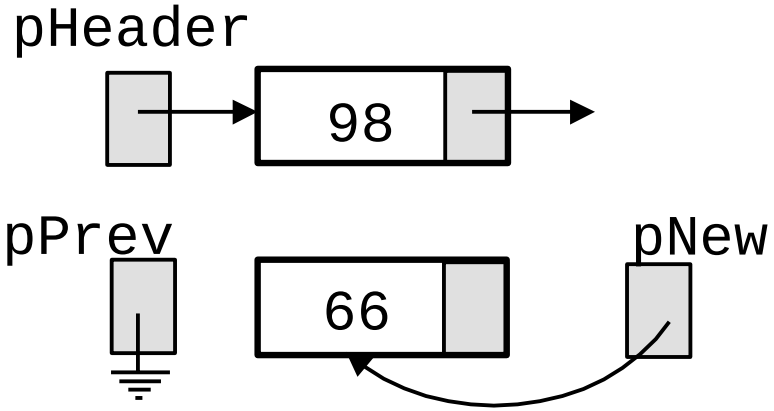

```
void MakeNULL( NodeType **ppData )
{
  *ppData = NULL;
}
```

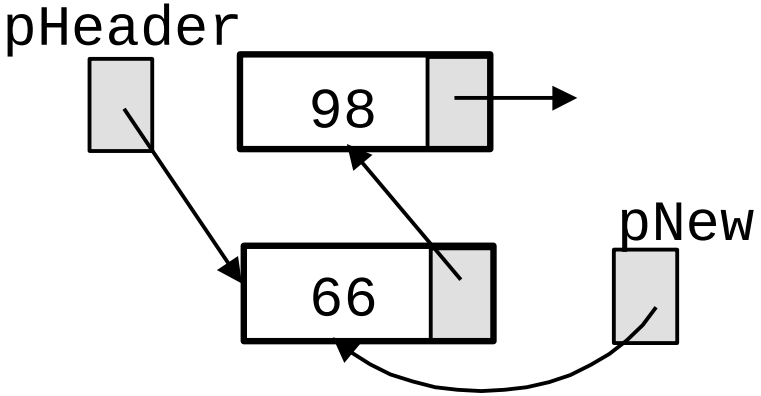
# Linked Lists: Insertion



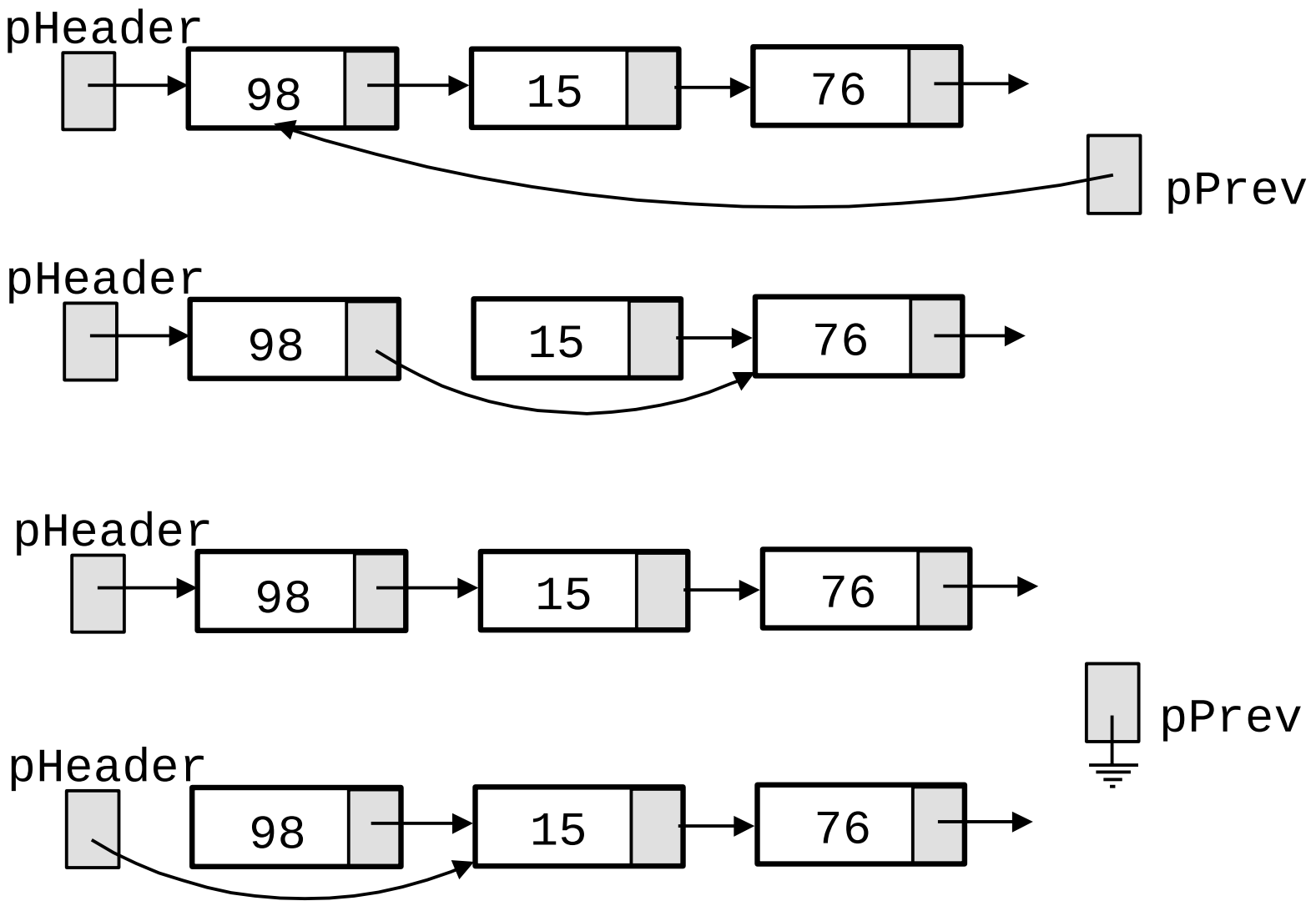



# Linked Lists: InsertNodeAfter

```
void InsertNodeAfter( NodeType **ppHeader,
                      NodeType *pPrev,
                      NodeType *pNew )
{
  if ( pPrev == NULL ) {
    pNew->next = *ppHeader;
    *ppHeader  =  pNew;
  } else {
    pNew->next  = pPrev->next;
    pPrev->next = pNew;
  }
}
```

# Linked Lists: Deletion



# Linked Lists: DeleteNodeAfter

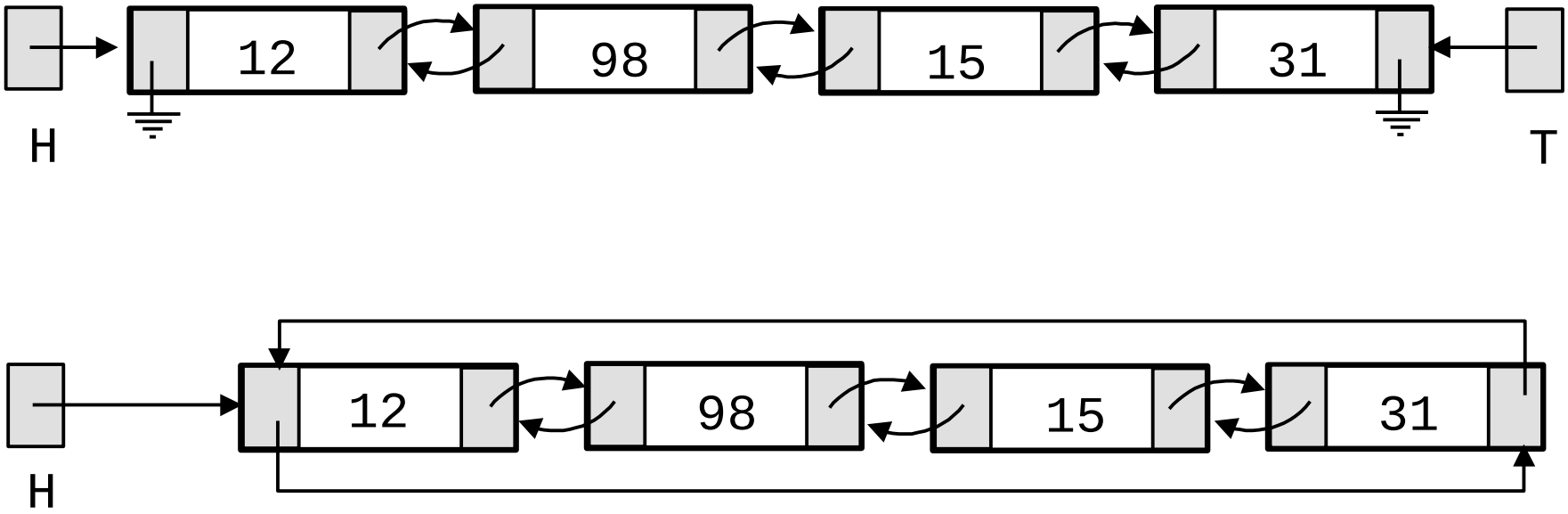
```
void DeleteNodeAfter( NodeType *pHeader,
                      NodeType *pPrev )
{
  NodeType   *pNode;

  if ( pPrev == NULL ) {
    pNode   = pHeader;
       pHeader = pHeader->next;
  } else {
    pNode       = pPrev->next;
    pPrev->next = pPrev->next->next;
  }
  free( pNode );
}
```

# Operations on Lists

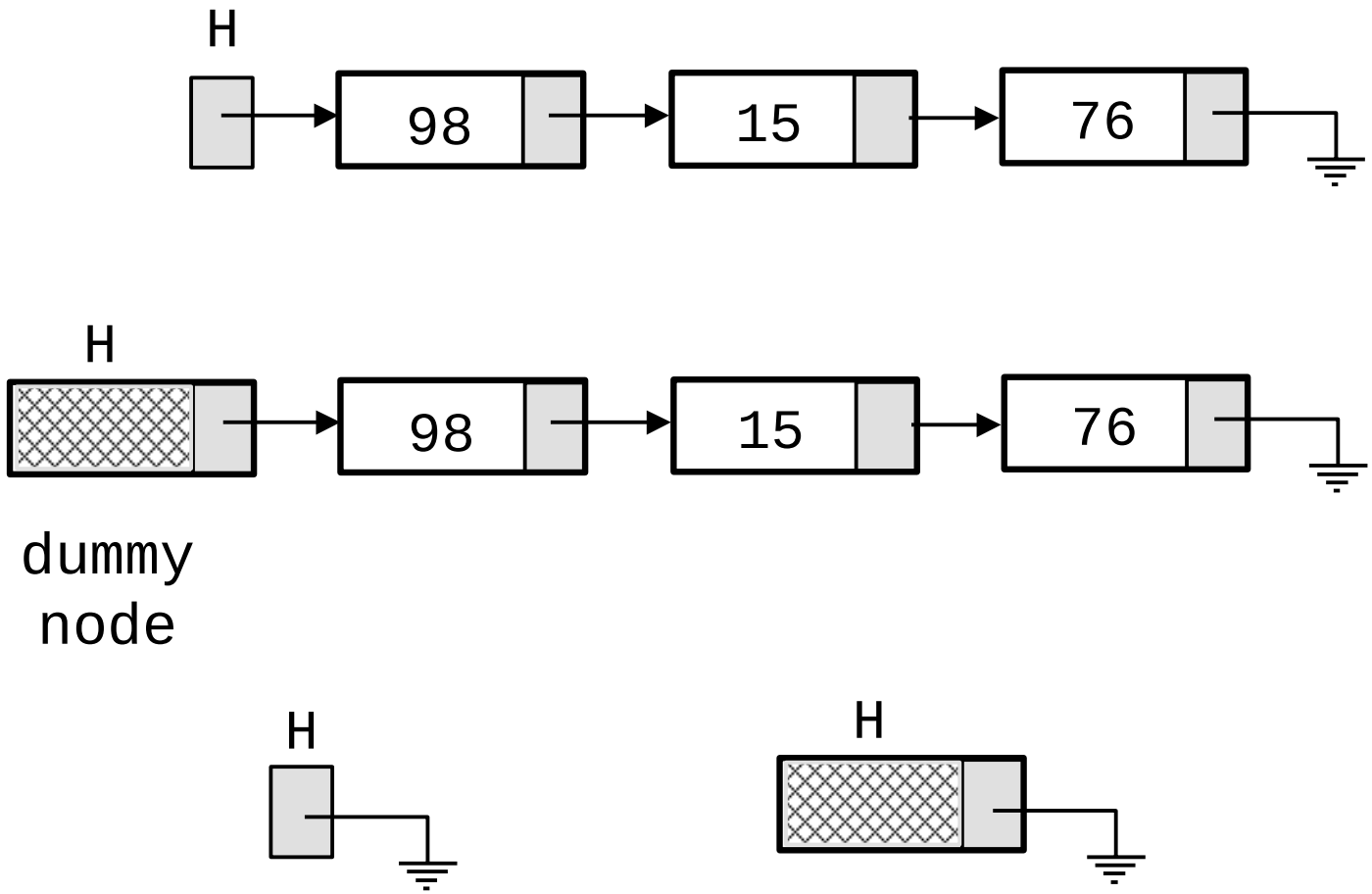
```
void InitializeList( ListType *pList );
Boolean IsFirst( ListType *pList, int window );
Boolean IsLast(  ListType *pList, int window );

void Start( ListType *pList, int *window );
void Next(  ListType *pList, int *window );
void Prev(  ListType *pList, int *window );
void Last(  ListType *pList, int *window );

void Delete(       ListType *pList, int *window );
void InsertAfter(  ListType *pList, int *window,
                   ItemType item );
void InsertBefore( ListType *pList, int *window,
                   ItemType item );
void Replace(      ListType *pList, int window,
                   ItemType item );
void Retrieve(     ListType *pList, int window,
                   ItemType *pItem );
```

S. Prasitjutrakul 1994

# Doubly Linked Lists



# Dummy Nodes

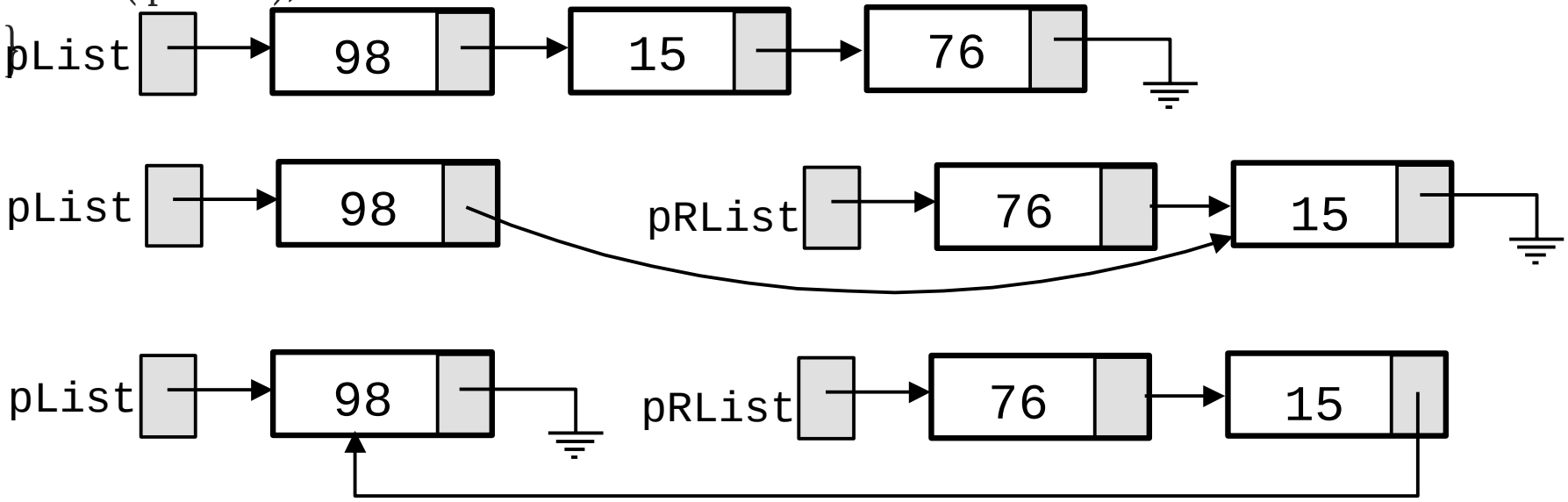


# Linked List: ListLength

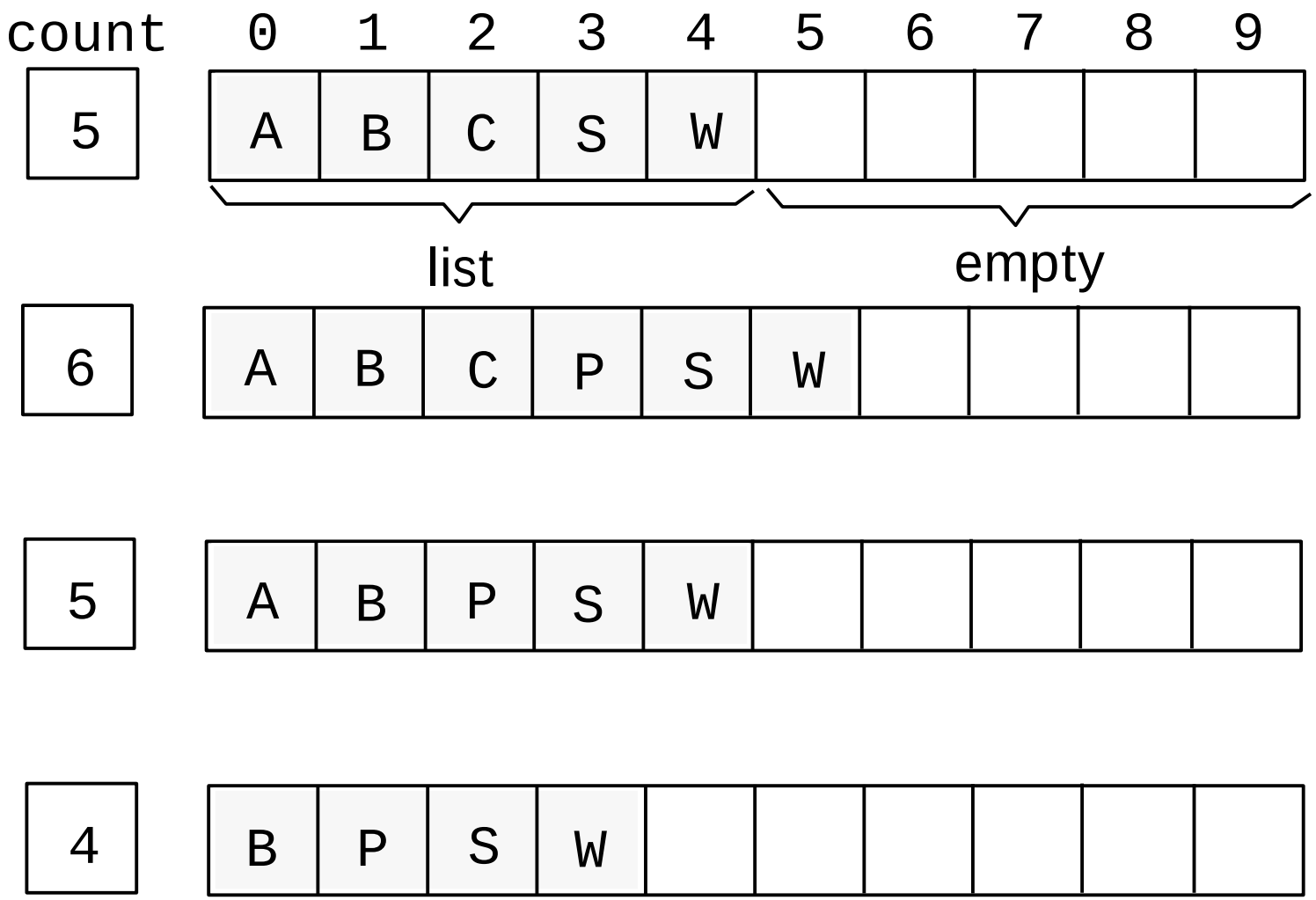
```
int  ListLength( NodeType *pHeader )
{
  return( pHeader == NULL ?
          0 : ( 1 + ListLength( pHeader->next ) ) );
}

int  ListLength( NodeType *pHeader )
{
  int      Length = 0;

  while ( pHeader != NULL ) {
    Length++;
    pHeader = pHeader->next;
 }
 return( Length );
}
```

# Linked Lists: ReverseList

```
NodeType *ReverseList( NodeType *pList )
{
  if ( pList->next == NULL ) return( pList );
  pRList = ReverseList( pList->next );
 pList->next->next = pList;
 pList->next = NULL;
 return( pRList );
}
```



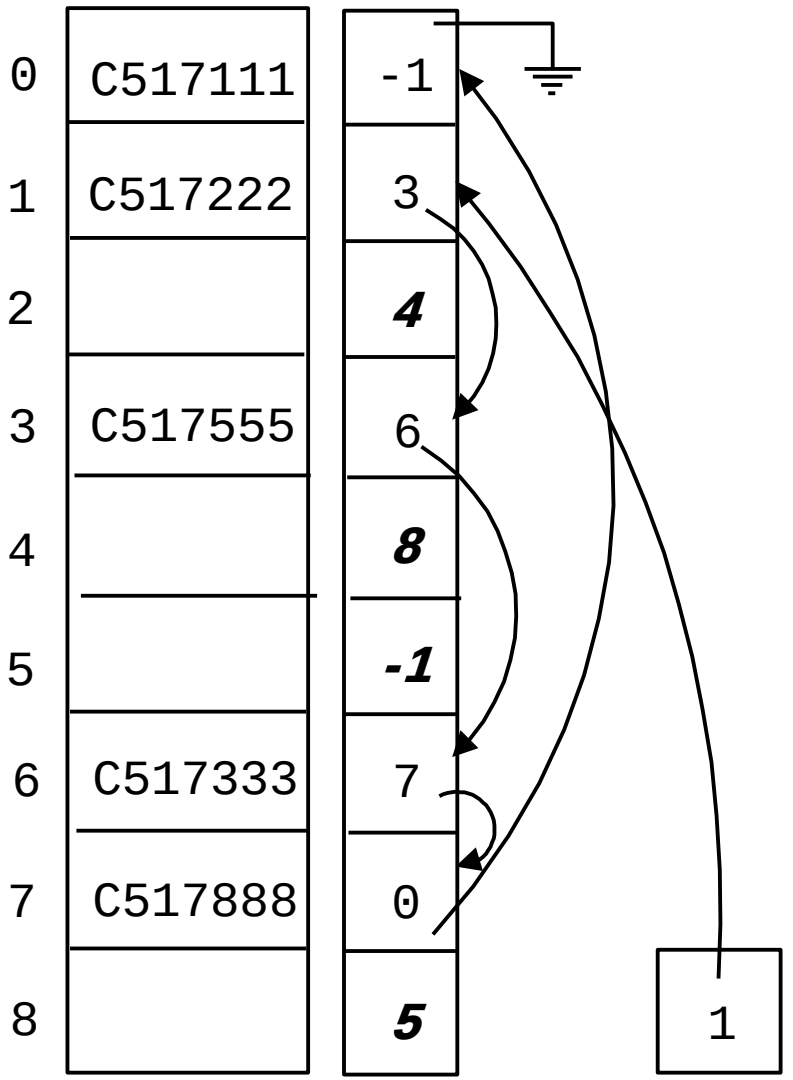
# Array Implementation of Lists

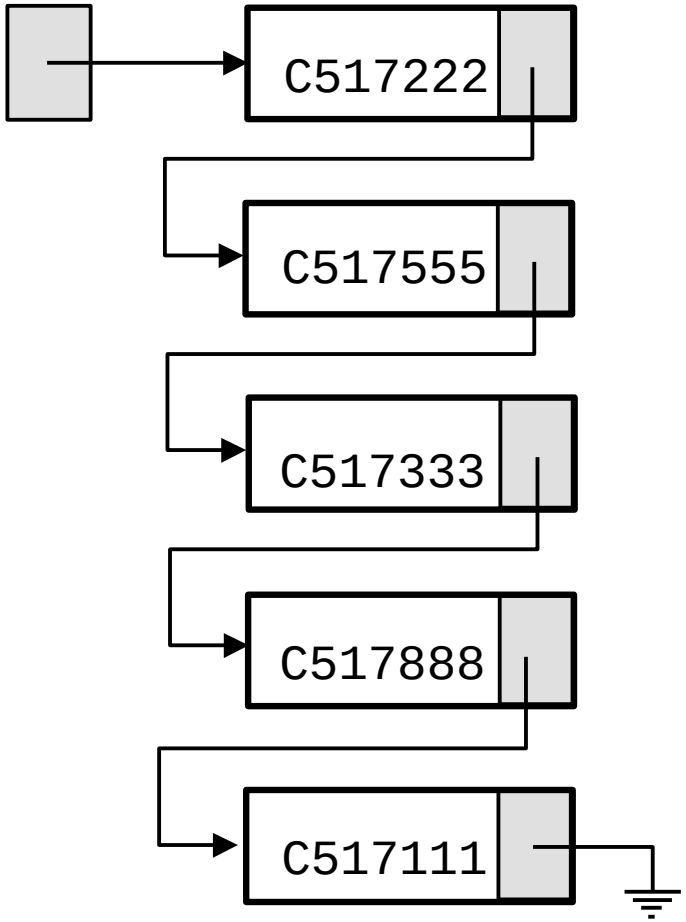


# Array Implementation of Lists

```
#define MAXLIST     10

typedef char        ItemType;
typedef enum BooleanTag { FALSE, TRUE } Boolean;
typedef struct NodeTag {
    int      count;
    ItemType   entry[MAXLIST];
} ListType;


int Size( ListType *pList )
{
 return( pList->count );
}


Boolean  ListIsEmpty( ListType *pList )
```

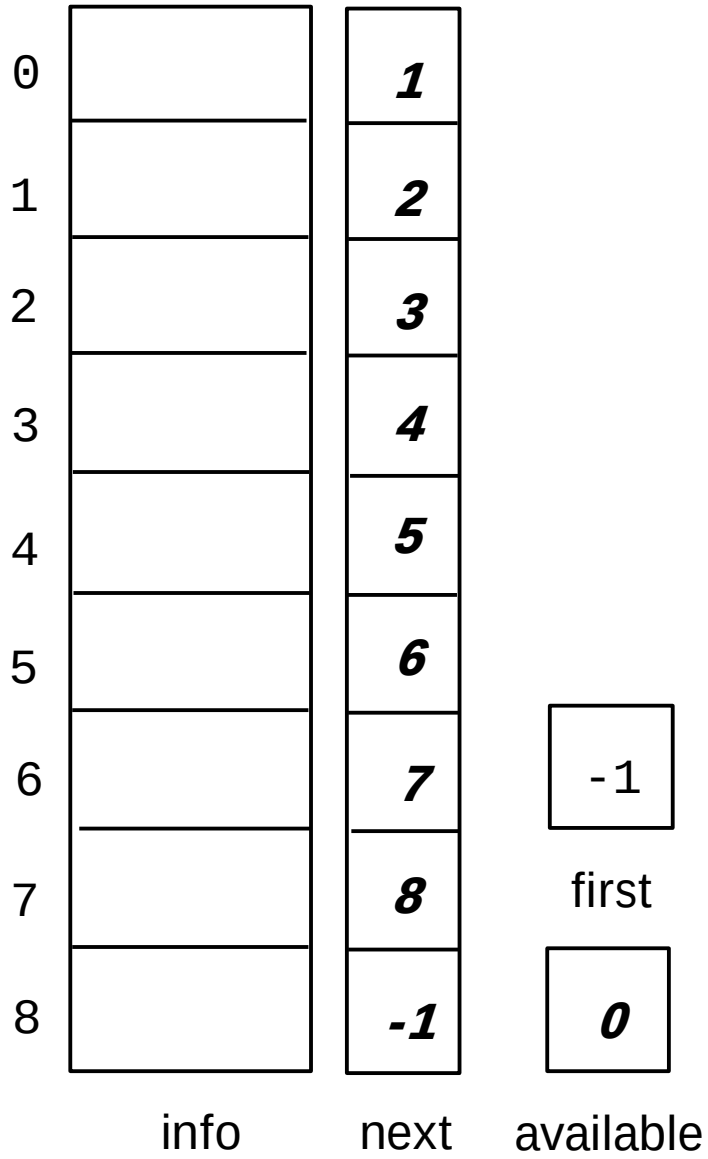
# Linked Lists in Arrays
## ( Cursor-Based Implementation of Lists )

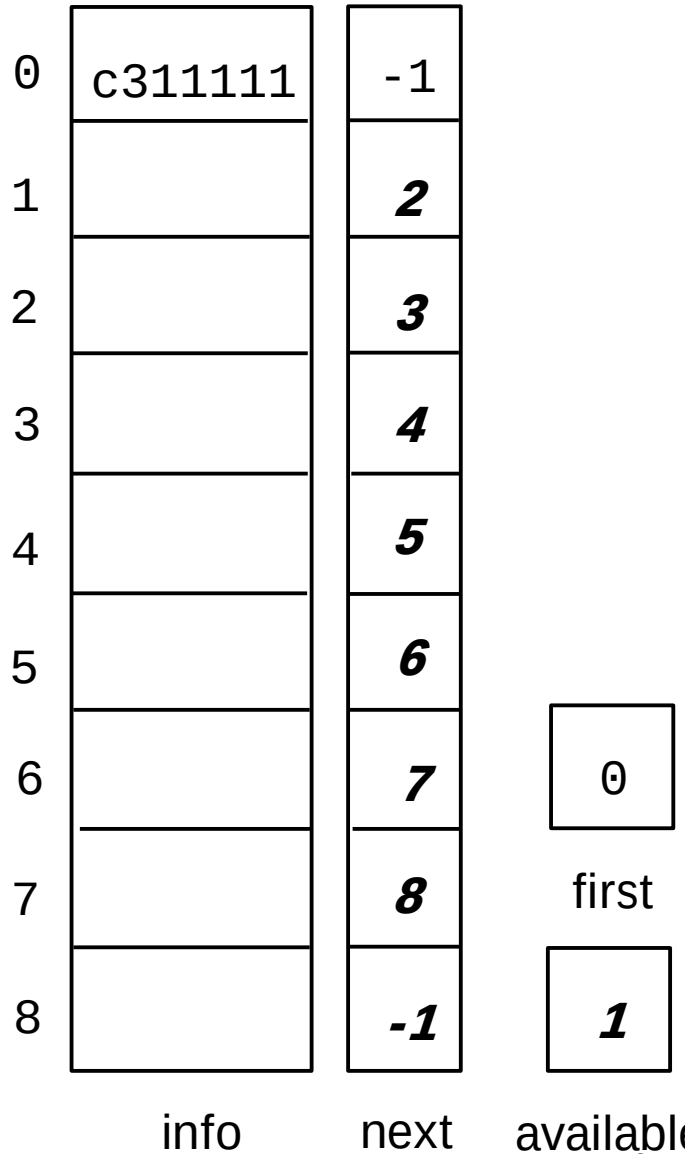


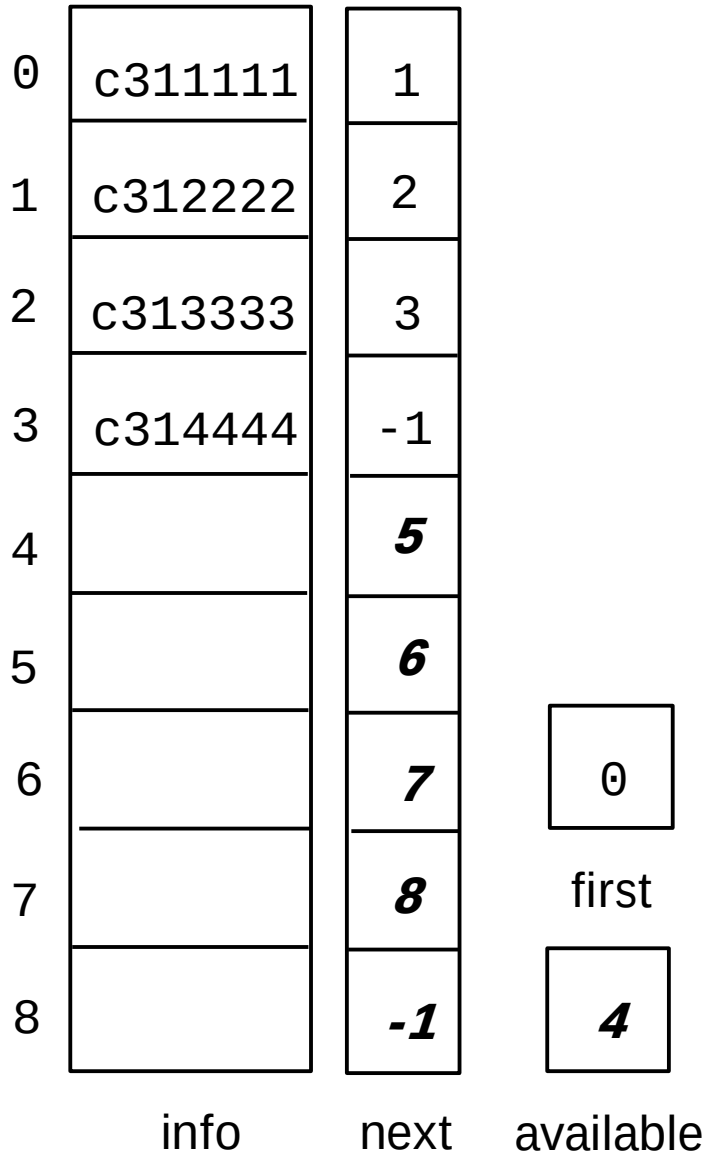
# Linked Lists in Arrays

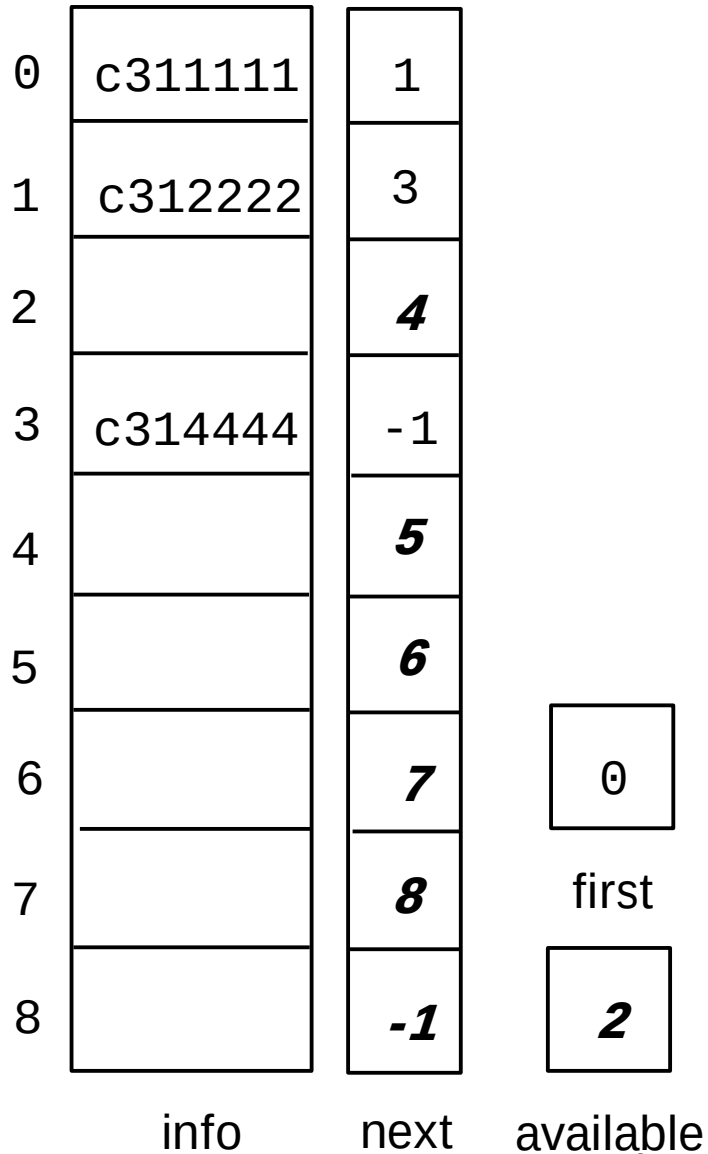
## ( Cursor-Based Implementation of Lists )



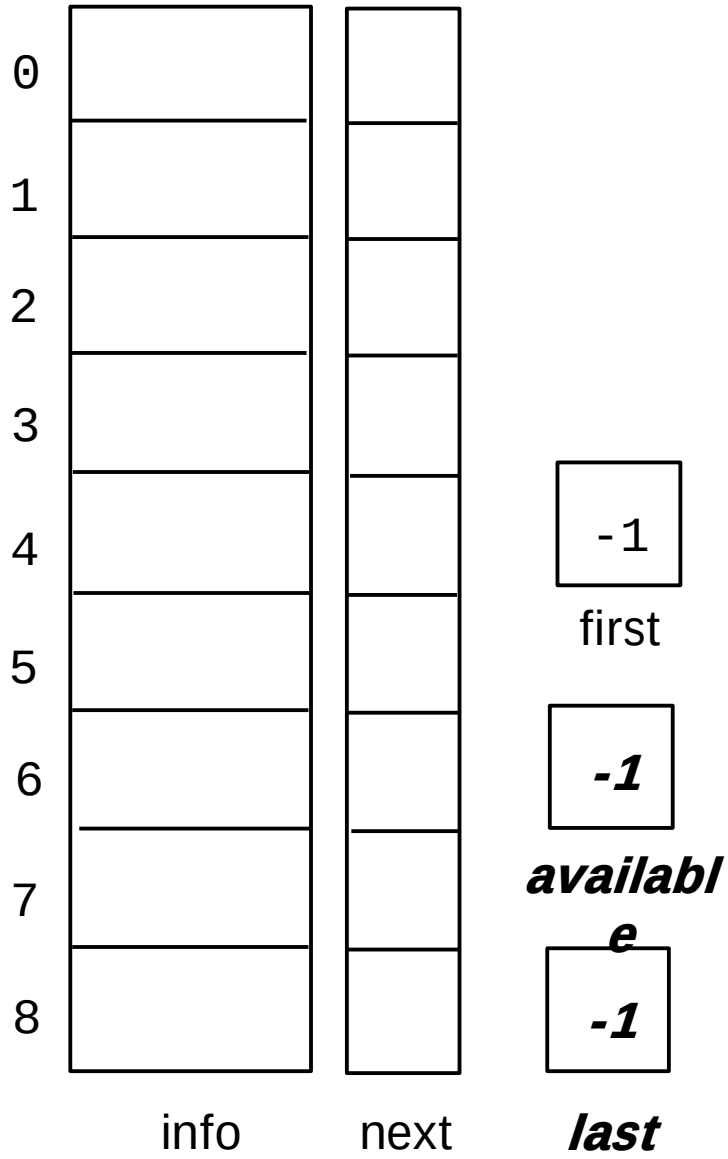



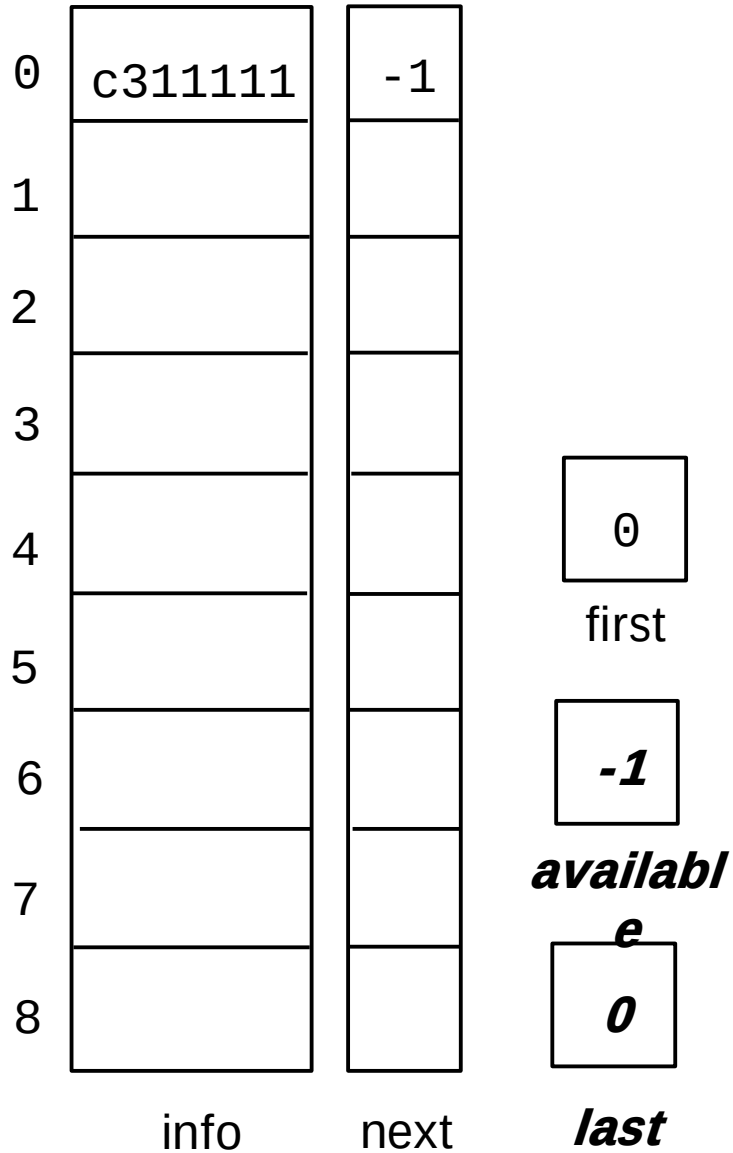
# Linked Lists in Arrays
## ( Cursor-Based Implementation of Lists )



# Linked Lists in Arrays
## ( Cursor-Based Implementation of Lists )



# Linked Lists in Arrays
## ( Cursor-Based Implementation of Lists )