Chunk List

By Daniel Szelogowski © 2017

Note: A full implementation can be found at
https://github.com/danielathome19/Chunk-List

Outline

- Discussion About Chunk Lists
  - What is a Chunk List?
  - Where is a Chunk List used?
- Implementation Details
  - Construction
  - Multithreading Methods
  - Index-Based Methods
  - Index Issues
  - Adding Elements
  - Removing Elements
  - Searching
  - Chunk Resizing
  - Sorting
- Complexity Analysis

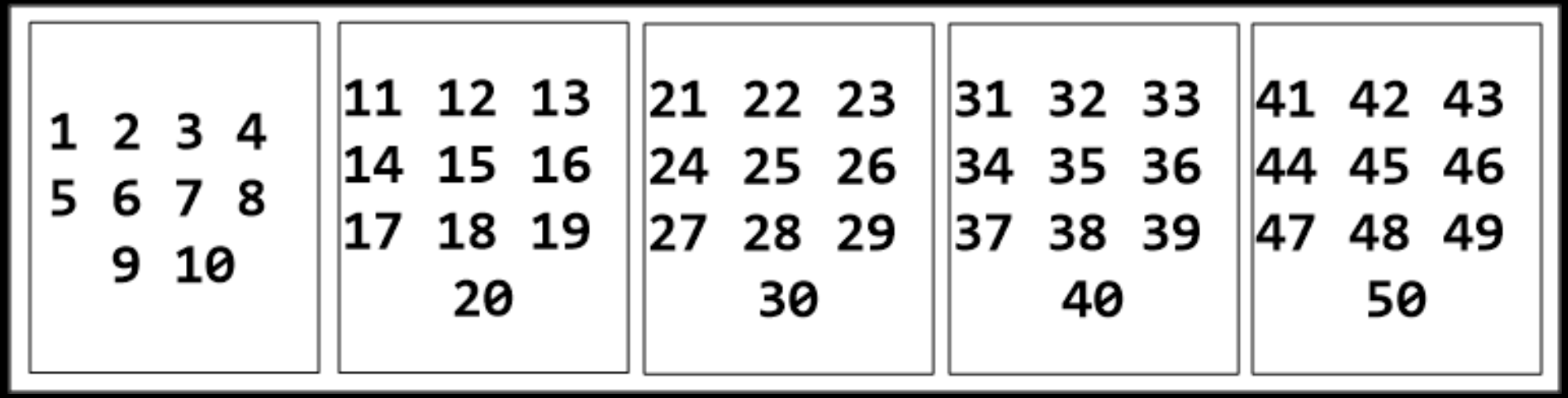
Discussion

What is a Chunk List?

- A chunk list is an array-based list of elements in which data is stored in inner lists of a certain capacity, allowing for easily modifiable and faster runtimes based on the number of elements being stored.
- A simple way to conceptualize a chunk list would be an ArrayList (dynamic array) of ArrayLists. The main list would contain the “chunks”, or ArrayLists that are not allowed to be filled past a specific capacity.
- Any time a “chunk” has reached capacity, a new ArrayList is added and items are added to that chunk from thereon.
- By doing this process and splitting our list into chunks, we can use parallel processing to our advantage. Using concurrency, we can run each chunk on a separate thread when doing tasks such as searching or removing.

Visual Example

- A chunk list containing the numbers 1 – 50 where the chunk size is set to 10 elements



Where Are Chunk Lists Used?

- Useful for storing very large and very small amounts of elements.
- Benefits shine especially when list is unsorted:
  - Fast searching
  - Fast removal
  - Fast insertion
- Implementation is easy and short.
- Sorting is quick even with large amounts of chunks.
- In any scenario, a chunk list can be used in place of an ArrayList especially, as well as something such as a Binary Search Tree, as searching may be faster based on processing power.
- Timed testing results typically yield to be approximately 3.5x faster than an ArrayList on average.

Implementation

Notes

- All code examples given are in the C# language.
- Examples are given for each main method the class should have implemented.
- Time complexities are listed for each method.
- A chunk size of $1/10^{th}$ the amount of elements being stored is generally ideal for speed and efficiency.

Construction

- The basis of the chunk list is the inner list. This is best implemented using some sort of dynamic list, such as ArrayList (or List in C#).
- This inner list will start out with a single list on the inside.
- Constructor must include an integer, chunk size. Otherwise, revert to a default size.
- New lists (chunks) will only be added to the main list when the chunk at the end has reached capacity.
- A chunk list may be implemented with generics (or templates) so long as the generic type is comparable.

Class Body Example

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

class ChunkList<T> where T : IComparable
{
    private List<List<T>> myList;
    private int chunkSize;

    private const int DEFAULT_SIZE = 1000;

    public ChunkList() : this(DEFAULT_SIZE)
    {
    }

    public ChunkList(int chunkSize)
    {
        this.chunkSize = chunkSize;
        myList = new List<List<T>>();
    }
}
```

Multithreading Methods

- Multithreading is an especially important part of chunk list implementation, as the basis of the list's speed is primarily the result of concurrency.
- For most methods in a chunk list, a new thread can be created for each chunk to be iterated through.
- A good example of this lies within C#'s Parallel.ForEach method, which will be referred to for this type of operation.
- Thread synchronization is not required when iterating, however keeping track of the thread state is important in some instances.

Index-Based Methods

- Accessing or modifying an element at a specified index (such as get, set, or removeAt methods) is somewhat more complex than in a regular list.
- To get the chunk where the position would be located, divide the index by the chunk size and cast it to an integer.
- To get the position in the chunk where the index would be, use modulo on the index by the chunk size.
- $chunk = int(index \ / \ chunkSize)$
- $chunkPosition = index \ \% \ chunkSize$
- Access via list[chunk][chunkPosition] (Where list is the main list inside the class)

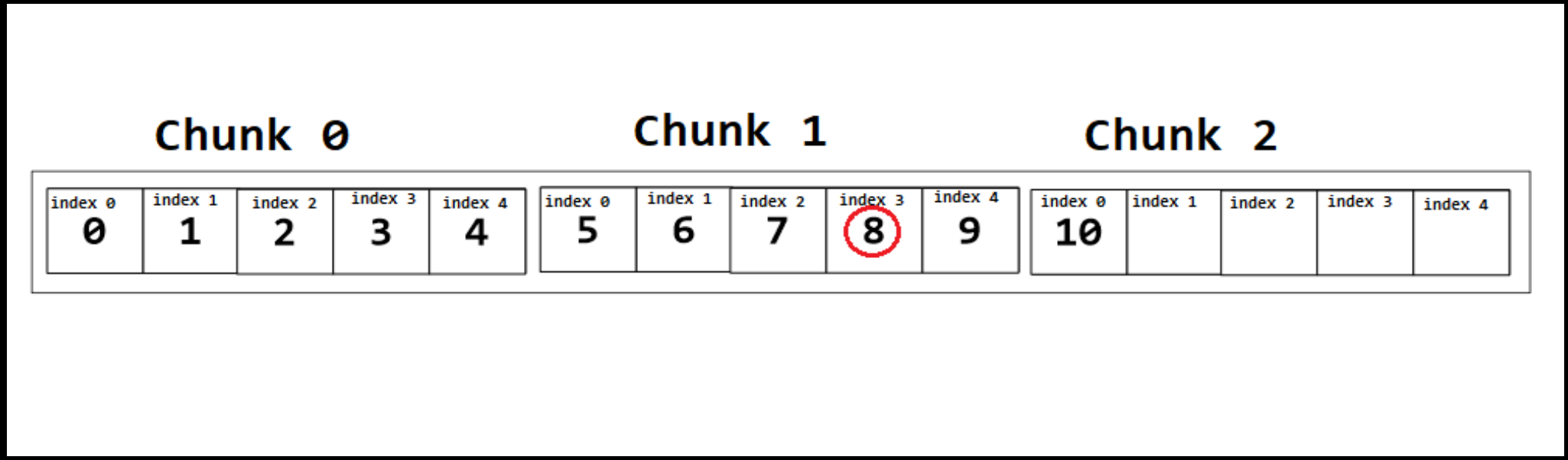
Index Accessing Example

```
private int convertIndexToChunk(int index)
{
    return index / chunkSize;
}

private int convertIndexToChunkPos(int index)
{
    return index % chunkSize;
}
```

Index Example

- The following example demonstrates accessing an element at index 8 in a chunk list containing numbers 0 – 10 with chunk size 5.
- Chunk: int(8 / 5) = 1
- Chunk Position: 8 % 5 = 3



Index Issues

- One issue with using indices in a chunk list, however, is the problem where items flow left within the chunk but do not migrate left from one to another if a chunk has an open slot. To implement so may hinder performance during removal.
- A very simple solution, however, would be to use recursion, such as within a try-catch statement with the index + 1.

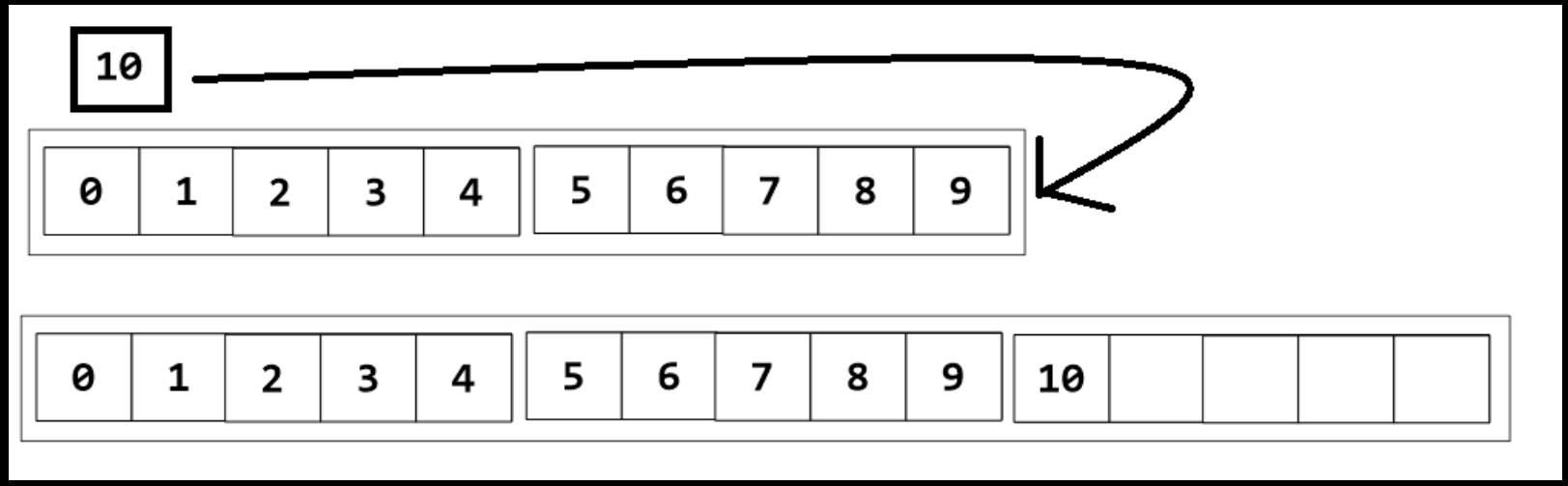
Index Issues Example

- The following example demonstrates a solution to the problem by counting up the index until an open position is found.

```
public T get(int index)
{
    if (index >= size()) throw new ArgumentOutOfRangeException();
    try
    {
        return myList[convertIndexToChunk(index)][convertIndexToChunkPos(index)];
    }
    catch (ArgumentOutOfRangeException)
    {
        return get(index + 1);
    }
}
```

Adding Elements

- Adding elements to a chunk list is simple; however, it does require that we check if each chunk is at capacity. Getting the size from the chunk should be Big-O (1), so this should not increase runtime marginally whatsoever.
- An element will naturally fall into the first open spot, or the first chunk that is not at capacity.
- If all chunks are at capacity, however, we need to add a new chunk to our list, then add the item it.



Adding Example

```
public void add(T t)
{
    foreach (List<T> currentList in myList)
    {
        if (currentList.Count != chunkSize)
        {
            currentList.Add(t);
            return;
        }
    }

    myList.Add(new List<T>());
    myList[myList.Count - 1].Add(t);
}
```

Removing Elements

- Removing elements is one of the fastest computational operations in a chunk list. This is where we can start using multithreading to our advantage.
- To remove an element, we can use a parallel for loop to concurrently check each chunk for the item.
- We can use a binary search to get the index that we're looking for.
- This is also where we need to be able to have access to the thread's state when we're looping through each chunk. If we only want to remove the first found instance of an element, we need to immediately break out of the parallel for loop.
- To remove all instances of an element within the list, we can still use a parallel for loop, and just call a removeAll method on each chunk.
- To clear the entire list, we can simply call clear on the main list (containing the chunks).

Removing Example

```
public void remove(T t)
{
    Parallel.ForEach(myList, (currentList, state) =>
    {
        int indx = currentList.BinarySearch(t);

        if (indx >= 0)
        {
            currentList.RemoveAt(indx);
            state.Break();
        }
    });
}
```

```
public void removeAll(T t)
{
    Parallel.ForEach(myList, (currentList) =>
    {
        currentList.RemoveAll(item => item.Equals(t));
    });
}
```

Searching

- Searching for an element is also where chunk lists shine.
- Once again we can use concurrency to get the shortest possible runtime, as now we can use a parallel for loop not only on the list itself, but on each chunk.
- Essentially, we can check most items in the list at the exact same time, meaning our runtime will be marginally smaller than using a linear search at worst case, and in the best case, a binary search.

Searching Example

```
public bool contains(T t)
{
    bool found = false;
    Parallel.ForEach(myList, (currentList, state) =>
    {
        Parallel.ForEach(currentList, (currentItem) =>
        {
            if (currentItem.Equals(t))
            {
                found = true;
                state.Break();
            }
        });
    });

    return found;
}
```

Chunk Resizing

- Should our data set grow marginally larger, we may need to resize our list.
- To do so however, means we'll need to rebalance our list, which is especially important if the chunk size we're changing to is smaller than the current one.
- We can make a temporary list containing all of our old items, change the chunk size, clear our old list, and then reflow our data back in.
- While somewhat costly performance-wise, this is an operation that should not be necessary to occur often.
- If the chunk size we want to adjust to is larger than the current one, however, we can simply leave the list as is and allow the elements to re-fill the chunks that are not yet at capacity.

Chunk Resizing Example

```
public void setChunkSize(int newChunkSize)
{
    if (newChunkSize > chunkSize)
    {
        chunkSize = newChunkSize;
    }
    else
    {
        var items = getList();
        chunkSize = newChunkSize;

        clear();

        foreach (var item in items)
        {
            add(item);
        }
    }
}
```

```
public List<T> getList()
{
    var items = new List<T>();

   foreach (var currentList in myList)
   {
       foreach (T currentItem in currentList)
       {
           items.Add(currentItem);
       }
   }

    return items;
}
```

Sorting

- Sorting our list is a fairly complex operation, similarly to searching.
- To properly sort our list, we do have to make a temporary list containing all elements of our chunk list. To do otherwise would only sort the chunks, which is not ideal as we do not know which order they will be inserted in.
- Using our temporary list, we can clear our main list and simply reflow all of our items back in after sorting it.

Sorting Example

- For this example, I simply used the search method implemented within C#'s List class, which follows the following rules:
  - If the partition size is fewer than 16 elements, it uses an insertion sort algorithm.
  - If the number of partitions exceeds 2 * log N, where N is the range of the input array, it uses a heapsort algorithm.
  - Otherwise, it uses a quicksort algorithm.

```
public void sort()
{
    var items = getList();

    items.Sort();

    clear();

    foreach (T item in items)
    {
        add(item);
    }
}
```

Complexity Analysis

Complexities – Basic Methods

- Complexities are listed with the following variables:
  - *C* being the number of chunks currently in the list.
  - *N* being the number of elements per chunk.
  - *P* being the number of processors.
  - *I* being the index input for the operation.

Operation	Average Case	Worst Case
Add	Θ(C)	N/A
Remove	Θ((log C * log N) / P)	Θ(log C * log N)
RemoveAll	Θ((log C * N) / P)	Θ(log C * N)
RemoveAt	Θ(1)	Θ(C * N - I)
Set	Θ(1)	Θ(C * N - I)
Get	Θ(1)	Θ(C * N - I)

Complexities – Additional Methods

- Complexities are listed with the following variables:
  - *C* being the number of chunks currently in the list.
  - *N* being the number of elements per chunk.
  - *P* being the number of processors.
  - *I* being the index input for the operation.

Operation	Average Case	Worst Case
GetList	Θ(C ^ 2 * N)	N/A
Contains (Search)	Θ((log C * log N) / P)	Θ(C * N)
Size (Count)	Θ(C)	N/A
SetChunkSize	Θ(1)	Θ(C ^ 2 * N)
Sort	Θ(C * N * log N)	Θ(C * N ^ 2)

The End.