copy is deleted. We repeat this copying and possible deletion of individual data units until our redundancy limit has been reached.

Blocks to Copy: Note that the span of an access requirement does not change by moving an interior data unit to another interior location. Cost can be reduced only by moving the blocks, i.e. data units, that are at the either ends of the access requirement. This observation greatly reduces the search space of data units to consider for copying. For the sake of simplicity of the algorithm, we operate on only one data block at a time.

Location to Copy: Based on the above observation, given an access requirement, we can possibly move the beginning or the end data units of an access requirement to its interior. This will reduce its span, thus reducing the EST for the layout. However, if the new location of the data unit is in the span of other access requirements, it increases the span of those accesses by one unit. We thus want to find a location in the span of the access requirement under consideration but is in the span of the least number of other access requirements. Such a location is identified using a simple linear search through the span of the access requirement.

Moving versus Copying: A data unit can be accessed by multiple access requirements. If that data unit is an extremal unit for an access requirements, and if it is moved to its interior, it affects span of those access requirements that use the same data unit. That is the main reason that we are copying and not moving these data units. By copying the data unit the other access requirements can still access it in its old location. The only increase in span for those access requirements would possibly be an extra unit resulting from the copying itself but it would not come from having to use the new copy. Nevertheless, if by using the new copy the span of one or more of the other access requirements reduces, then those access requirements should use the new copy instead of the old copy. If all the access requirements use the new copy so that the old data block is not used by any access, then it can be deleted. In this later case, we are really moving the data block.

Data Block processing order: We now need to figure out how to use this information to decide in what order the copying should be done. When considering the order we are only considering cases where we are copying and not the cases where we are moving. This is because copying adds a space cost so we need a way to decide in what order it should be done whereas moving does not have that cost. For each data unit, its total benefit is the amount that is reduces the total seek time (EST). For a given data unit to be copied to a specified location, let k_i be the benefit to access requirement i that is attached to the data unit. We will say that $k_i = 0$ if the access requirement will use the old copy and not have its span increased by the addition of a new copy, $k_i = -1$ if the access requirement will use the old copy but still have its span increased, and $k_i > 0$ if the access requirement will use the new copy. Let I be the set of access requirements that use the data unit. Let J be the set of access requirements not in I whose span overlaps our data units. We can now describe the benefit, the reduction in seek time, as follows:

$$Benefit = -\Delta EST = \sum_{i \in I} k_i - |J|.$$

Before doing any actual copying, we compute the above described benefit for each start and end data unit for each access requirement. We store all the benefits into a binary search tree, i.e. a heap, sorted in descending order by benefit amount. That way we will easily be able to choose the data unit that provides the most benefit in expected seek time. We will also make a special list L of cases where a data unit is copied and then can be deleted, because by doing that, all the access requirements will benefit.

Because doing the moving for the list L does not increase the storage, we will first go through that list and perform those moves. After each of the moves we have to recompute the costs and update the tree and list. Once L is empty we will take the data unit in the binary search tree that provides the most benefit and perform the copy. We will then recompute L and the binary search tree. We will continue to do those steps until we have run out of available space for redundancy. As a summary, the psuedo-code of this algorithm is shown as algorithm 1.

```
Start with the data units which each with at least one access
requirement (AR);
Initialize AR, heap H, and list L;
for each data unit do
    Find number of overlapping access requirements and store the
    number with the data unit;
end
for each AR P's head and tail data units U do
    make old copy list L' empty;
    if U is head data unit then
        Let S = data unit in P after U;
    else
        Let S = data unit in P before U;
    end
    Set BENEFIT=distance(S,U);
    Let U' be the potential copy of U;
    Search data units between S and U for min number of
    overlapping ARs and put U' there;
    make old copy list L', the list that stores the ARs that will use
    the old data unit, empty;
    for each AR T that also uses U do
        if T will be shorter by using U' then
            Add T's benefit to BENEFIT;
        else
            Add T to list L';
        end
    end
    Add BENEFIT to heap H;
    if L' is empty then
        add U to list L;
    end
end
while BREAK has not been called do
    while L is not empty do
        Take out random element and move the data unit;
        Update heap H and list L;
    end
    if there is more space for redundancy then
        pop best element U from H;
        copy U to its destination;
        update nodes in H and entries in L for affected access
        requirements;
        update H and L;
    else
        call BREAK
    end
end
```

Algorithm 1: Pseudo-code for our algorithm

4 Run-time and Storage Analysis

We now analyze the running time and storage requirements of our algorithm. We will denote N as the number of data units and A as