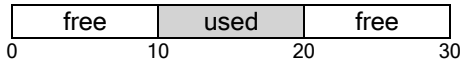
17

Boş Alan Yönetimi (Free-Space Management)

Bu bölümde, ister malloc kütüphanesi (işlemin öbeğinin sayfalarını yönetme) ister işletim sisteminin kendisi (işlemin adres alanının yönetim bölümleri) olsun, herhangi bir bellek yönetim sisteminin temel bir yönünü tartışmak için bellek sanallaştırmayla ilgili tartışmamızdan ufak bir sapma yapacağız. Özellikle de, **boş alan yönetimini(free-space management)** çevreleyen konuları tartışacağız.

Sorunu daha spesifik bir hale getirelim. **Çağrı (paging)** kavramını tartışırken göreceğimiz gibi, boş alan yönetimi muhakkak kolay olabilir. Yönettiğiniz alan sabit boyutlu birimlere bölündüğünde kolaydır; böyle bir durumda, bu sabit boyutlu birimlerin listesini tutmanız yeterlidir; bir kullanıcı bunlardan birini istediğinde, ilkgirişe döndürmemiz gerekmektedir.

Boş alan yönetiminin daha zor (ve ilginç) hale geldiği yer yönettiğimiz boş alanın değişken boyutlu birimlerden oluşmasıdır; Bu kullanıcı düzeyinde bir bellek ayırma kütüphanesi (malloc() , free() olduğu gibi) ve sanal belleği gerçekleştirmek için **bölümleme (segmentation)** kullanıldığında fiziksel belleği yöneten bir işletim sisteminde ortaya çıkar. Her iki durumda da, mevcut sorun **dış parçalanma (external segmentation)** olarak bilinir. Boş alan farklı boyutlarda küçük parçalara bölünür ve parçalanır; dahasonraki istekler başarısız olabilir çünkü toplam boş alan miktarı isteğin boyutunu aşsa bile isteği karşılayabilecek tek bir bitişik alan yoktur.



Yukarıdaki şekil bu sorunun bir örneğini göstermektedir. Bu durumda, toplam boş alan 20 byte'tır; nr yazık ki, her biri 10'luk iki parçaya bölünmüş durumda. Sonuçta, 20 byte boş olsa dahi 15 byte'lık bir istek başarısız olur. Böylece bu bölümde değindiğimiz soruna ulaşmış oluyoruz.

PÜF NOKTA: Boş Alan Nasıl Yönetilir
Değişken boyutlu istekleri karşılarken boş alan nasıl yönetilmelidir? Parçalanmayı en aza indirmek için hangi stratejiler kullanılabilir? Alternatif yaklaşımların zaman ve mekan ek yükleri nelerdir?

17.1 Varsayımlar

Bu tartışmanın çoğu kullanıcı düzeyinde bellek ayırma kütüphanelerinde bulunan ayırıcıların muhteşem geçmişine odaklanacaktır. Bunun için de Wilson'ın mükemmel anketinden [W+95] faydalanıyoruz ancak daha fazla ayrıntı için ilgili okuyucuları kaynak belgenin kendisini incelemeye teşvik ediyoruz.

Malloc() ve free() gibi temel bir arabirim tarafından sağlandığını varsayıyoruz. Özellikle, `void *malloc(size_t size)` uygulamanın talep ettiği byte sayısı olan tek bir parameter alır. O boyuttaki (veya daha büyük) bir bölgeye işaretçiyi (belirli bir türde olmayan veya C dilinde **boş işaretçi (void pointer)**) geri verir. Tamamlayıcı rutin `void free(void *ptr)` bir işaretçi alır ve karşılık gelen öbeği serbest bırakır.. Arayüzün kastettiğine dikkat edin: kullanıcı, alanı boşaltırken kütüphaneyi boyut hakkında bilgilendirmez bu nedenle kütüphane, kendisine yalnızca bir işaretçi verildiğinde bellek yığınının ne kadar büyük olduğunu anlayabilmelidir. Biraz sonra bunu bu bölümde nasıl yapacağımızı tartışacağız

Bu kitaplığın yönettiği alan,tarihsel olarak yığın olarak bilinir ve yığındaki boş alanı yönetmek için kullanılan genel veri yapısı bir tür serbest listedir(free list). Bu yapı, belleğin yönetilen bölgesindeki tüm boş alan yığınlarına referanslar içerir. Tabii ki, bu veri yapısının kendi basşına bir liste olması gerekmez, ancak boş alanı izlemek için sadece bir tür veri yapısı olması gerekir.

Ayrıca, yukarıda da bahsedildiği üzere, öncelikli olarak **dış parçalanma (external fragmentation)** ile ilgilendiğimizi varsayıyoruz. Ayırıcılar, elbette **dahili parçalanma (internal fragmentation)** sorunu da yaşayabilirler. Eğer bir ayırıcı talep edilenden daha büyük bellek yığınları dağıtırsa, böyle bir yığındaki sorulmamış (and dolayısıyla kullanılmayan) herhangi bir alan, dahili parçalanma olarak kabul edilir (çünkü atık, tahsis edilen birimin içinde gerçekleşir) ve bu da alan israfına bir örnektir. Bununla birlikte, kolaylık açısından ve iki tür parçalanma arasında daha ilginç olduğu için çoğunlukla dış parçalanmaya odaklanacağız.

Ayrıca bellek kullanıcıya dağıtıldıktan sonra, bellekte başka bir konuma taşınamayacağını da varsayacağız. Örneğin, bir program malloc()'u çağırırsa ve yığın içindeki bir boşluğa bir işaretçi verilirse, bu bellek bölgesi program free() çağrısı yoluyla geri dönünceye kadar esas olarak program tarafından kullanılmaktadır (ve kütüphane tarafınfan taşınamaz). Böylece, boş alanın **sıkıştırılması (compaction)** mümkün değildir. Hangisi

80 sayfa uzunluğundadır; Bu yüzden gerçekten ilgilenmelisiniz!

Parçalanmayla mücadelede yararlı olacaktır. Bununla birlikte, işletim sisteminde **segmentasyon (bölümleme)** uygulanırken parçalanma ile başa çıkmak için kullanılabilir (segmentasyonla ilgili bahsedilen bölümde tartışıldığı gibi).

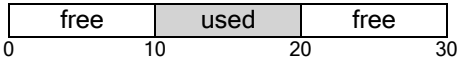
Son olarak, ayırıcının bitişik bir byte bölgesini yönettiğini varsayacağız. Bazı durumlarda, bir ayırıcı o bölgenin büyümesini isteyebilir. Örneğin, kullanıcı düzeyinde bir bellek ayırma kütüphanesi, alanı dolduğunda yığını büyütmek için çekirdeğe çağrı yapabilir (sbrk gibi bir system çağrısı vasıtasıyla) . Bununla birlikte, daha kolay olması açısından bölgenin ömrü boyunca tek bir sabit boyutta olduğunu varsayacağız.

17.2 Düşük Seviye Mekanizmalar

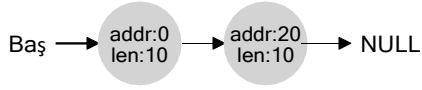
Bazı ilke ayrıntılarına girmeden önce, çoğu ayırıcıda kullanılan bazı ortak mekanizmaları ele alacağız. İlk olarak, herhangi bir ayırıcıdaki ortak tenkiklerden olan bölme ve birleştirmenin temellerini tartışacağız. Daha sonra, tahsis edilen bölgelerin nasıl hızlı ve nispeten kolay bir şekilde takip edilebileceğini göstereceğiz. Son olarak, neyin boş olup olmadığını takip etmek için boş alanın içerisinde basit bir listenin nasıl oluşturulacağını tartışacağız.

(Bölme Ve Birleştirme) Splitting and Coalescing

Boş bir liste yığında kalan boş alanı tanımlayan bir dizi öğe içerir. O yüzden, aşağıdaki 30 byte'lık yığını varsayalım:



Bu yığın için boş listede iki öğe olacaktır. Bir giriş ilk 10 byte'lık boş bölümü (0-9 byte) ve diğer bir giriş ise öteki boş bölümü (20-29 byte) açıklar:

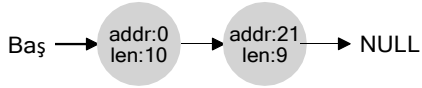


Yukarıda açıklandığı gibi, 10 byte'tan daha büyük herhangi bir istek başarısız olur (Null döndürür). Bu boyutta kullanılabilir tek bir bitişik bellek parçası yoktur. Ancak istek 10 byte'tan daha küçük bir şey içinse ne olur)

Yalnızca tek bir byte bellek talebimiz olduğunu varsayalım. Bu durumda ayırıcı, **bölme (splitting)** olarak bilinen bir işlem gerçekleştirir. Bu İsteği karşılayabilecek

Bir C programında bellek yığınına bir işaretçi verdiğinizde, diğer değişkenlerde veya yürütme sırasında belirli bir noktada kayıtlarda depolanabilecek bölgeye yapılan tüm referansları (işaretçiler) belirlemek zordur. Bu daha güçlü yazılmış çöp toplayan dillerde böyle olmayabiliyor. Bu da parçalanmayla mücadele için bir teknik olarak sıkıştırmayı mümkün kılıyor.

boş bir bellek parçası bulur ve onu ikiye böler. İlk yığın çağıran kişiye geri dönecektir; ikinci yığın ise listede kalacaktır. Bu nedenle, yukarıdaki örneğimizde 1 byte için bir talepte bulunursa ve ayırıcı bu talebi karşılamak için listedeki iki öğeden ikincisini kullanmaya karar verirse, Malloc() çağrısı 20'ye döndürür (tahsis edilen 1 byte'lık bölgenin adresi) ve liste şu şekilde görünür:



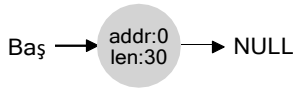
Resimde, listenin temelde bozulmadan kaldığını görebilirsiniz. Tek değişiklik, boş bölgenin artık 20 yerinde 21'den başlaması ve bu gölgenin uzunluğunun artık 9 olmasıdır. Bu nedenle, istekler belirli bir boş öbeğin boyutundan daha küçük olduğunda ayrıma genellikle ayırıcılarda kullanılır.

Boş alanın **birleştirilmesi (coalescing)** pek çok ayırıcıda doğal bir mekanizma olarak bilinir. Örneğimizi bir kez daha ele aldığımızda (Boş bir 10 byte, kullanılan bir 10 byte ve bir diğer boş 10 byte).

Bu (küçük) yığın göz önüne alındığında, bir uygulama free(10) öğesini çağırdığında ve böylece yığının ortasındaki boşluğu döndürdüğünde ne olur ? Eğer bu boş alanı çok fazla düşünmeden listemize geri eklersek, aşağıdakine benzer bir liste elde edebiliriz:

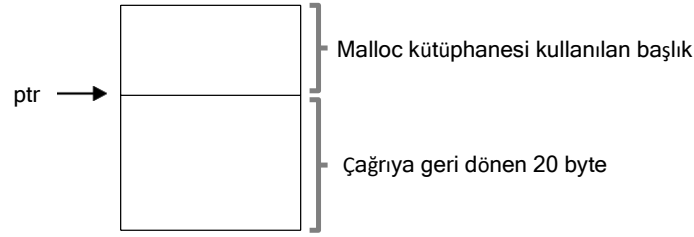
Soruna dikkat edelim: Yığının tamamı artık boş olsa da, görünüşe göre her biri 10 byte'lık üç parçaya bölünmüş durumda. Bu nedenle, bir kullanıcı 20 byte talep ederse basit bir liste geçişi bu kadar boş bir öbeği bulamayacak ve sonuç olarak başarısızlıkla sonuçlanacaktır.

Ayırıcıların bu sorunu önlemek için yaptığı şey, bir yığın bellek serbest bırakıldığında bu boş alanı birleştirmektir: Aslında mantık basit: bellekte boş bir yığın döndürürken, döndürdüğünüz yığının adreslerine ve yakındaki boş alan yığınlarına dikkatlice bakın. Yani boşaltılan alan bir (ya da bu örnekte olduğu gibi iki) mevcut boş yığının hemen yanında bulunuyorsa, bunları daha büyük tek bir boş yığın halinde birleştirin. Böylece birleştirme sonrasında ile nihai listemiz aşağıdaki gibidir.

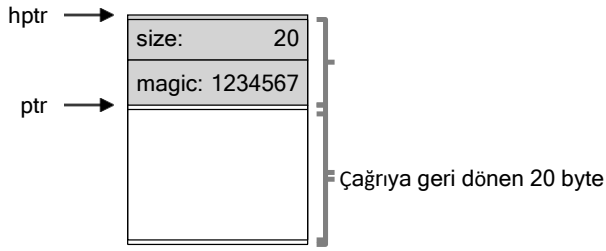


Gerçekten de, herhangi bir ayırma yapılmadan önce yığın listesi ilk başta böyle görünüyordu. Birleştirme ile bir ayırıcı. Uygulama için geniş boş uzantıların kullanılabilir olmasını daha iyi sağlayabilir.

[3]This discussion assumes that there are no headers, an unrealistic but simplifying assumption we make for now.



Şekil 17.1: **Tahsis Edilmiş Bölge + Başlık (An Allocated Region Plus Header)**



Şekil 17.2: **Başlığın Özel İçeriği (Specific Contents Of The Header)**

Tahsis Edilen Bölgelerin Büyüklüğünün Takibi

Free (void*ptr) arabiriminin bir boyut parametresi almadığını farketmiş olabilirsiniz. Bu nedenle, bir işaretçi verildiğinde malloc kütüphanesinin serbest bırakılan bellek bölgesinin boyutunu hızlı bir şekilde belirleyebileceği ve böylece alanı tekrar boş listeye dahil edebileceği varsayılır.

Bu görevi tamamlamak için, Ayırıcıların çoğu genellikle dağıtılan bellek öbeğinden hemen önce, bellekte tutulan bir **başlık (header)** bloğunda fazladan bilgi depolar. Örneğe tekrar bakalım (Şekil 17.1). Bu örnekte, ptr ile gösterilen, 20 byte boyutunda ayrılmış bir bloğu inceliyoruz; kullanıcının malloc()' u çağırdığını ve sonuçları ptr'da sakladığını düşünün, yani, `ptr = malloc(20);`.

Başlık, minimum tahsis edilen bölgenin boyutunu içerir (bu durumda 20); ayrıca ayrıştırmayı hızlandırmak için ek işaretçiler, ek bütünlüğü sağlamak için sihirli bir sayı ve öbür bilgileri içerebilir. Bölgenin boyutunu ve bunun gibi bir sihirli sayı içeren basit bir başlık olduğunu varsayalım:

```
typedef struct {
    int size;
    int magic;
} header_t;
```

Yukarıdaki örnek, şekil 17.2'de gördüğünüz gibi görünecektir. Kullanıcı free (ptr) öğesini çağırdığında, kütüphane başlığın nereden başladığını bulmak için basit bir işaretçi aritmetiği kullanır:

```
void free(void *ptr) {
    header_t *hptr = (header_t *) ptr - 1;
    ...
```

Başlığa yönelik böyle bir işaretçi elde ettikten sonra, kütüphane sihirli sayının mantıksal kontrol olarak beklenen değerle eşleşip eşleşmediğini kolayca belirleyebilir (`assert(hptr->magic == 1234567)`) ve boş bırakılan yeni bölgenin toplam boyutunu şu şekilde basit matematik yoluyla hesaplayabilir (yani başlığın boyutunu bölgenin boyutuna ekleyerek). Son cümledeki ufak ama önemli ayrıntıya dikkat edin: boş bölgenin boyutu, başlığın boyutu ile kullanıcıya ayrılan alanın boyutunun toplamıdır. Bu nedenle, bir kullanıcı N byte'lık bellek talep ettiğinde, kütüphane N boyutunda boş bir yığın aramaz. Bunun yerine, başlığın ve N'in boyutunda boş bir yığın arar.

Boş Listeye Katma (Embedding A Free List)

Şimdiye kadar boş listemizi basit kavramsal bir varlık olarak ele aldık; Bu sadece yığındaki boş bellek parçalarını açıklayan bir listedir. Ama boş alanın içinde böyle bir listeyi nasıl oluştururuz?

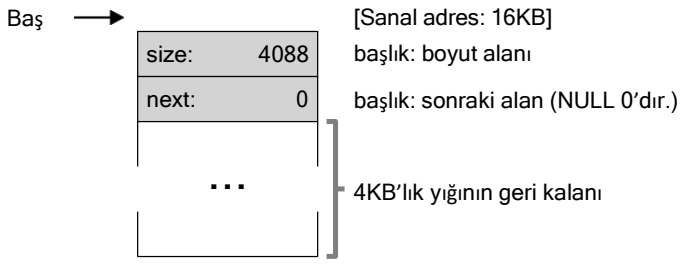
Daha tipik bir listede, yeni bir düğüm ayırırken, bu düğüm için alana ihtiyacınız olduğunda malloc() kütüphanesini çağırırsınız. Ne yazık ki, bunu bellek ayırma kütüphanesinde yapamazsınız. Bunun yerinde listeyi boş bir alanın içerisinde oluşturmanız gerekmektedir. Kulağa biraz garip gelse de endişelenmeyin, bu yapılamayacak kadar tuhaf bir şey değil!

Yönetilecek 4096 byte'lık bir bellek yığınımız olduğunu varsayalım (yani yığın 4kb'tır). Bunu boş bir liste olarak yönetmek için, önce söz konusu listeyi başlatmamız gerekmektedir. Başlangıçta listenin 4096 boyutunda (başlık boyutu hariç) bir girişi olmalıdır. Aşağıda listenin bir düğümünün açıklaması verilmiştir.

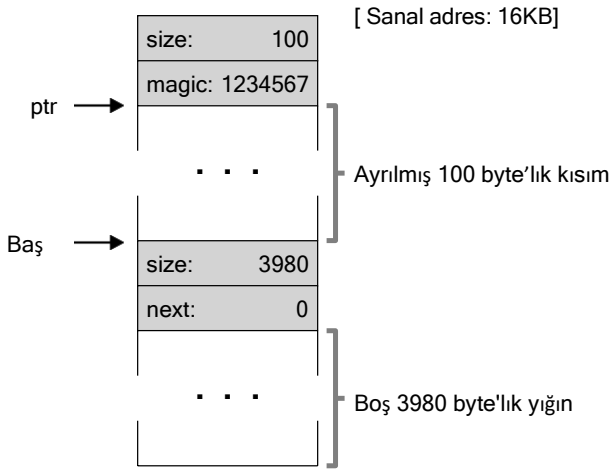
```
typedef struct___node_t {
    int              size;
    struct___node_t *next;
} node_t;
```

Şimdi ise yığını başlatan ve boş listenin ilk öğesini o alana yerleştiren bir koda bakalım. Yığının mmap() adı verilen bir system çağrısı yoluyla elde edilen bazı boş alanlar içerisinde inşa edildiğini varsayıyoruz. Elbette böyle bir yığını inşa etmenin tek yolu bu değil ama bu örnek de bizim işimizi görüyor. İşte kod:

```
// mmap() boş alan yığınına bir işaretçi döndürür.
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```



Şekil 17.3: **Boş Yığınlı Bir Yapı (A Heap With One Free Chunk)**

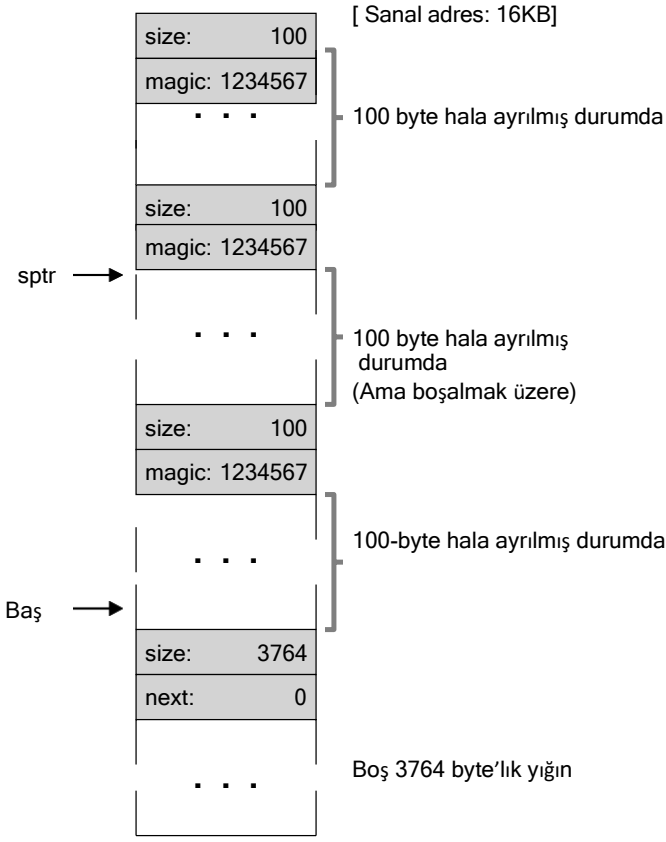


Şekil 17.4: **Yığın: Bir Ayırmadan Sonra (A Heap: After One Allocation)**

Bu kodu çalıştırdıktan sonra listenin durumu, 4088'lik boyutta tek bir girişe sahip olmasıdır. Bu küçük bir yığın ama bizim için güzel bir örnek teşkil ediyor. Baş işaretçi, bu aralığın başlangıç adresini içermektedir; şimdi bu adresin 16kb olduğunu varsayalım (ancak herhangi bir sanal adres iyi olacaktır). Böylece yığın Şekil 17.3 'teki gibi görünür.

Şimdi, varsayalımki 100 byte boyutunda bir bellek yığını isteniyor. Bu isteğe hizmet vermek için, kütüphane önce isteği karşılayacak kadar büyük bir yığın bulacaktır; çünkü yalnızca bir boş parça bulunduğundan (boyut: 4088) bu yığın seçilecektir. Daha sonra yığın ikiye bölünecektir: bir yığın, isteğe hizmet edecek kadar büyük (yukarıdaki başlıkta açıklandığı gibi), ve geriye kalan boş alan. 8 byte'lık bir başlığı (bir tamsayı boyutu ve bir sihirli tamsayı boyutu), yığındaki boşluk artık şekil 17.4'te gördündüğü gibi olur.

Böylece, 100 byte'lık istek üzerine, kütüphane mevcut bir boş yığından

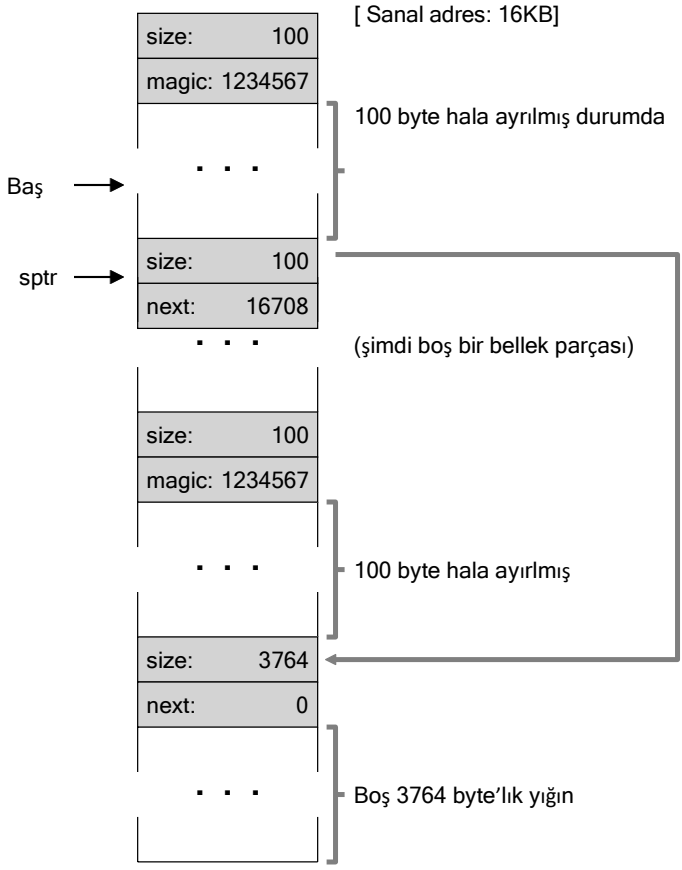


Şekil 17.5: **Üç Parçaya Ayrılmış Boş Alan (Free Space With Three Chunks Allocated)**

108 byte alan ayırır ve ona bir işaretçi döndürür (yukarıdaki şekilde sptr ile gösterilmiştir), başlık bilgisini, daha sonra free() üzerinde kullanılmak üzere tahsis edilen alanın hemen öncesinde depolar ve listedeki boş bir düğümü 3980 byte'a azaltır (4088 eksi 108).

Şimdi, her biri 100 byte'lık üç tahsis edilmiş bölge olduğunda yığına bakalım (bu yığının göresel hali şekil 17.5'te gösterilmiştir).

Burada görebileceğiniz gibi, yığının ilk 324 byte'lık kısmı tahsis edilmiştir, böylece o alanda üç başlık ve çağıran program tarafından kullanılan üç tane 100 byte'lık bölge görüyoruz. Boş liste sıkıcılığını korumaya devam ediyor: yalnızca tek bir düğüm (baş tarafından işaret edilen) ancak şimdi üç bölmeden

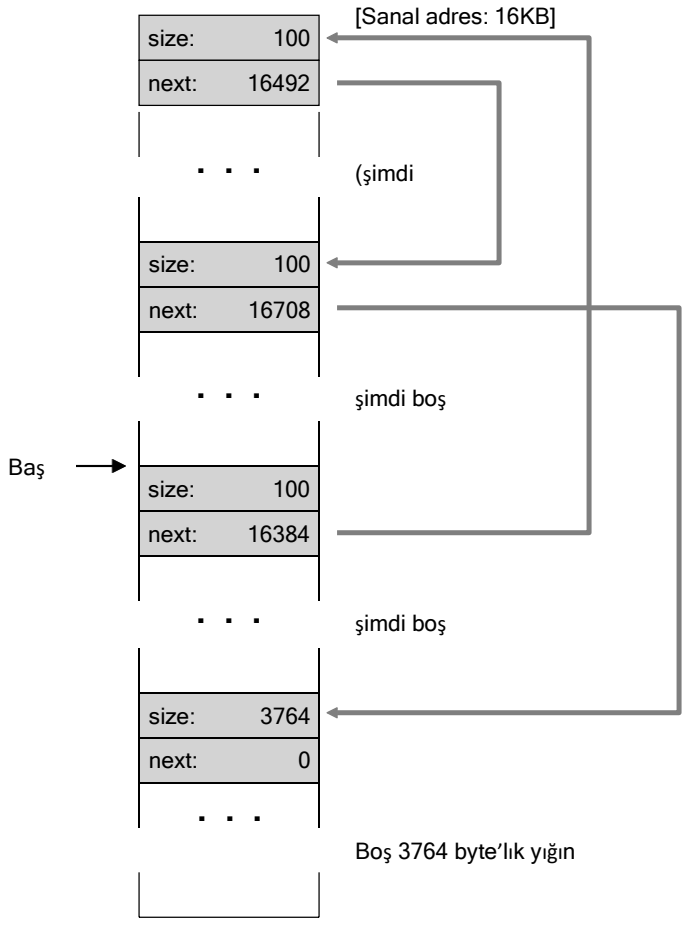


Şekil 17.6: **İki Parçaya Ayrılmış Boş Alan(Free Space With Two Chunks Allocated)**

Yalnızca 3764 byte boyutunda. Ancak, çağıran program free() aracılığıyla bir miktar bellek döndürdüğünde ne olur?

Bu örnekte uygulama, free(16500) öğesini çağırarak ayrılan belleğin orta yığınını döndürür (16500 değerine, bellek bölgesinin başlangıcı olan 16384, önceki parçanın 108'ine ve bu yığın için başlığın 8 byte'ına eklenerek ulaşılır). Bu değer önceki diyagramda sptr işaretçisi tarafından gösterilmiştir.

Kütüphane boş bölgenin boyutunu hesaplar ve ardından boş yığını tekrar boş listeye ekler, Boş listenin başına yerleştirdiğimizi varsayarsak, alan şuna benzer (şekil 17.6).



Şekil 17.7: **Birleştirilmemiş Serbest Liste (A Non-Coalesced Free List)**

Şimdi küçük boş yığın (listenin başında gösterilen 100 byte) ve büyük boş yığın (3764 byte) ile başlayan bir listemiz var. Nihayet listemizde artık birden fazla öğre var ve evet boş alan parçalanmış, talihsiz ama sık rastalanan bir olay.

Son bir örnek: şimdi kullanımda olan son iki parçanın serbest kaldığını varsayalım. Birleşme olmazsa parçalanma olur (şekil 17.7).

Şekilde de görebileceğiniz gibi, büyük bir karmaşıklık yaşıyoruz! niçin? basit, listeyi **birleştirmeyi (coalesce)** unuttuk. Hafızanın tamamı boş olmasına rağmen, parçalara ayrılmıştır. Böylece bir olmamasına rağmen parçalanmış bir hafıza gibi görünür. Çözüm basit: listeyi gözden geçirin ve komşu parçaları **birleştirin (merge)** bu işlem bittiğinde yığın tekrar bütün olacaktır.

Yığını Büyütme (Growing The Heap)

Pek çok ayırma kütüphanesinde bulunan son bir mekanizmayı ele almalıyız. Özellikle yığında boş yer kalmazsa ne yapmalıyız? En basit yaklaşım sadece başarısız olmaktır. Bazı durumlarda bu tek seçenektir ve bu yüzden NULL değerini döndürmek onurlu bir yaklaşımdır. Kendini kötü hissetme en azından denedin ve başarısız olmana rağmen iyi mücadele ettin.

Geleneksel ayırıcıların çoğu küçük boyutlu bir yığınla başlar ve bittiğinde işletim sisteminden daha fazla bellek talep eder. Tipik olarak bu , yığını büyütmek için bir tür sistem çağrısı (örneğin, çoğu UNIX sisteminde sbrk) yaptıkları ve ardından yeni parçaları oradan tahsis ettikleri anlamına gelir. Işletim sistemi, sbrk isteğine hizmet vermek için boş fiziksel sayfaları bulur, bunları istekte bulunan işlemin adres alanına eşler ve ardından yeni yığının sonuncu değerini döndürür. Bu noktada, daha büyük bir yığın kullanılabilir ve istek başarıyla yerine getirilebilir.

17.3 Temel Stratejiler

Artık elimizde bazı makineler olduğuna göre, boş alanı yönetmek için bazı temel stratejilerin üzerinden geçelim. Bu yaklaşımlar çoğunlukla kendi başınıza düşünebileceğiniz oldukça basit ilkelere dayanmaktadır. Okumadan önce deneyin ve tüm alternatifleri (veya bazı yenilikleri) bulup bulamayacağına bakın

Ideal ayırıcı hem hızlıdır hem de parçalanmayı en aza indirir. Ne yazık ki, tahsis akışı boş istekler gelişi güzel olabileceğinden (sonuçta bunlar programcı tarafından belirlenir), yanlış girdi seti verildiğinde herhangi bir belirli strateji oldukça kötü iş çıkarabilir. Bu nedenle, 'en iyi' yaklaşımı tarif etmeyeceğiz, bunun yerine bazı temel hususlardan bahsedeceğiz ve bunların artılarını ve eksilerini tartışacağız

En Uygun Uyum (Best Fit)

En uygun uyum (Best Fit) stratejisi oldukça basittir: ilk olarak, boş listede arama yapın ve istenen boyuttan büyük veya daha büyük olan boş bellek parçalarını bulun. Ardından, o aday grubundan en küçük olanı döndürün bu sözde uygun parçadır (en küçük uyum olarak da adlandırılabilir). Boş hisseden bir kez geçmek, döndürülecek doğru bloğu bulmak için yeterlidir.

En iyi uyumun ardındaki sezgi basittir: kullanıcının istediğine yakın bir blog döndürerek en uygun boşa harcanan alanı azaltmaya çalışır. Ancak bunun bir bedeli vardır. Sade uygulamalar, doğru boş blok için kapsamlı bir arama gerçekleştirirken ağır bir performans cezası öder.

En kötü Uyum (Worst Fit)

En kötü uyum yaklaşımı, en uygun yaklaşımın tersidir; en büyük parçayı bulun ve istenen miktarı iade edin; Kalan (büyük) parçayı ise boş listede tutun. Böylece en kötü uyum, en uygun yaklaşımdan doğabilecek çok sayıda

küçük parça yerine büyük parçaları serbest bırakmaya çalışır. Ancak bir kez daha, tam bir boş alan araması gereklidir ve bu nedenle bu yaklaşım maliyetli olabilir. Daha da kötüsü çoğu araştırma, kötü performans gösterdiğini ve bunun da yüksek genel giderlere sahipken aşırı parçalanmaya yol açtığını gösteriyor.

İlk Uyum (First Fit)

Ilk uyum (First Fit) yöntemi, yeterince büyük olan ilk bloğu bulur ve istenen miktarı kullanıcıya döndürür. Daha önce olduğu gibi, kalan boş alan sonraki istekler için boş tutulur.

Ilk uyumun hız avantajı vardır — tüm boş alanların kapsamlı bir şekilde aranması gerekmez — ancak bazen boş listenin başlangıcını küçük nesnelerle kirletir. Böylece, ayırdığınız serbest liste sırasını nasıl yönettiği bir sorun haline gelir. bir yaklaşım **adrese dayalı sıralamayı (adressed-based ordering)** kullanmaktır; listeyi boş alanı adresine göre sıralayarak birleştirme daha kolay hale gelir ve parçalanma azalma eğilimi gösterir.

Sonraki Uyum (Next Fit)

Ilk uyum aramasını her zaman listenin başında başlatmak yerine **sonraki uyum (Next Fit)** algoritması, listede en son bakılan konumda fazladan bir işaretçi tutar Amaç boş alan arabalarını liste boyunca daha düzgün bir şekilde yaymak ve böylece listenin başındaki parçalamayı önlemektir. Kapsamlı bir aramadan bir kez daha kaçınıldığı için böyle bir yaklaşımın performansı ilk uyum stratejisine oldukça benzerdir.

Örnekler (Examples)

İşte yukarıdaki stratejileri birkaç örnek. 10, 30 ve 20 boyutlarında, üzerinde 3 öğe bulunan boş bir liste hayal edin (burada başlıkları ve diğer ayrıntıları göz ardı edeceğiz, bunun yerine sadece stratejilerini nasıl çalıştığını odaklanacağız):

15 boyutlu bir tahsis talebi olduğunu varsayalım. En uygun yaklaşım, tüm listeyi arar ve isteği karşılayabilecek en en küçük boş alan olduğu için 20'nin en uygun yer olduğunu bulur:

Bu örnekte olduğu gibi ve genellikle en uygun yaklaşımda olduğu gibi, artık küçük bir boş yığın kalmıştır En kötü uyum yaklaşımı buna benzerdir ancak bunun yerine bu örnekte 30 olan en büyük parçayı bulur. Işte sonuç listesi:

Bu örnekteki ilk uygun strateji, en kötü uyum stratejisi ile aynı şeyi yapar ve ayrıca talebi karşılayabilecek ilk serbest bloğu bulur. tek fark arama maliyetindedir. Hem En uygun muyum hem de en kötü huyum tüm listeye bakar. ilk uyum stratejisi, Uygun olanı bulana kadar yalnızca boş parçaları inceler böylece arama maliyetini düşürür.

bol örnekler tahsis ilkelerinin sadece yüzeyini çizer. Daha fazla detay için gerçek iş yükleri ve daha karmaşık ayırıcı davranışlar (örneğin, birleştirme) ile daha ayrıntılı analiz gereklidir. Belki ödev bölümü için bir şey söylersiniz?

17.4 Diğer Yaklaşımlar

Yukarıda açıklanan temel yaklaşımların ötesinde, bellek tahsisini bir şekilde iyileştirmek için önerilen bir dizi teknik ve algoritma vardır. Değerlendirmeniz için bunlardan bir kaçını burada listeliyoruz(yani, en uygun tahsisten biraz daha fazlasını düşünmenizi sağlamak için).

Ayrılmış Listeler (Segregated Lists)

Bir süredir ortalıkta dolaşan ilginç bir yaklaşım, **ayrılmış listelerin (Segregated Lists)** kullanılmasıdır. Temel gayet basittir. Belirli bir uygulamanın yaptığı popüler boyutta bir (veya birkaç) istek varsa, yalnızca o boyuttaki nesneleri yönetmek için ayrı bir liste tutulur. Diğer tüm istekler daha genel bir bellek ayırıcı ya iletilir.

Böyle bir yaklaşımın yararları nettir. Belirli bir boyuttaki istekler için ayrılmış bir bellek parçasına sahip olarak, parçalanma çok daha az endişe vericidir. Ayrıca, ayırma ve boş istekler doğru boyutta olduklarında karmaşık bir liste araması gerekmediğinden oldukça hızlı bir şekilde sunulabilir.

Her iyi fikir gibi, bu yaklaşım da sisteme yeni komplikasyonlar getirir. Örneğin, genel havuzun aksine belirli bir boyuttaki özel isteklere hizmet eden bellek havuzuna ne kadar bellek ayrılmalıdır? Belirli bir ayırıcı, übermühendis Jeff Bonwick'in (solaris çekirdeğinde kullanılmak üzere tasarlanmış) **levha ayırıcısı (slab allocator)** bu sorunu oldukça güzel bir şekilde ele alıyor.[B94].

Özellikle, çekirdek ön yüklendiğinde sık sık talep edilmesi muhtemel çekirdek nesneleri (şifreler, dosya sistemi düğümleri, vb.) için bir dizi nesne önbelleğine tahsis eder. Bu nedenle nesne ön belleklerin her biri, belirli bir boyuta ayrılmış boş listelerdir ve hızlı bir şekilde bellek tahsisi ve serbest istekler sunar. Belirli bir önbellekte boş alan azaldığında daha genel bir bellek ayırıcıdan bazı bellek dilimleri ister (istenen toplam miktar, sayfa boyutunun ve söz konusu nesnenin bir katıdır). Aksine, belirli bir levha içindeki nesnelerin referans sayıları sıfıra gittiğinde, genel ayrıcı bunları uzmanlaşmış ayrıcadan geri alabilir, bu genellikle VM sistemi daha fazla belleğe ihtiyaç duyduğunda yapılır.

YANİ: BÜYÜK MÜHENDİSLER GERÇEKTEN BÜYÜKTÜR

Jeff Bonwick gibi mühendisler (yalnızca burada bahsedilen levha ayırıcı yazmakla kalmayıp, aynı zamanda harika bir dosya sistemi olan ZFS'nin de lideriydi) silikon vadisi'nin kalbidir. Neredeyse her büyük ürünün veya teknolojinin arkasında yetenekleri ve bağlılıkları açısından ortalamanın çok üzerinde olan bir insan (veya küçük bir grup insan) vardır. Mark Zuckerberg'in dediği gibi: " rolünde olağanüstü olan biri, oldukça iyi olan birinden biraz daha iyi değil, 100 kat daha iyidir". Bu nedenle bugün hala bir veya birkaç kişi dünyanın çehresini sonsuza dek değiştirecek bir şirket kurabilir (Google, Apple veya Facebook gibi). Çok çalışırsanız, böyle bir "100x" kişi de olabilirsiniz. Aksi takdirde, böyle başarılı bir kişiyle çalışın çoğu kişinin bir ayda öğrendiğinden daha fazlasını bir günde öğreneceksiniz. Bunu başaramazsanız üzülebilirsiniz.

Levha ayırıcı ayrıca, listelerdeki boş nesneleri önceden başlatılmış bir durumda tutarak ayrılmış liste yaklaşımlarının çoğunun ötesine geçer. Bonwick, veri yapılarının başlatılmasının ve yok edilmesinin maliyetli olduğunu gösteriyor [B94]. Serbest bırakılan nesneleri belirli bir listede başlatılmış durumda tutarak levha ayırıcı nesne başına sık başlatma ve yok etme döngülerini önler ve böylece genel giderleri önemli ölçüde azaltır.

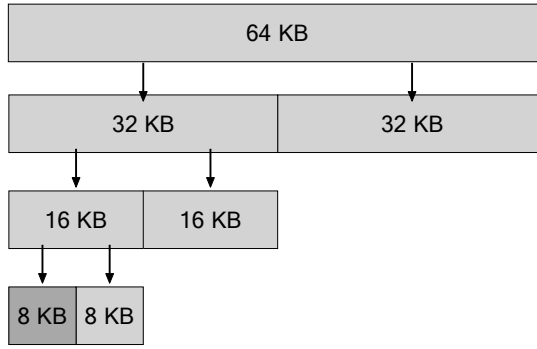
Arkadaş Ayırma (Buddy Allocation)

Birleştirme bir ayırıcı için kritik olduğundan birleştirme işlemini basit hale getirmek için bazı yaklaşımlar tasarlanmıştır. **Ikili arkadaş ayırıcı (binary buddy allocator)** o da iyi bir örnektir.

Böyle bir sistemde, boş bellek önce kavramsal olarak 2^N boyutunda büyük bir alan olarak farz edilir. Benlik için istek yapıldığında boş alan, isteği karşılayacak kadar büyük bir blok bulunana kadar boş alanı yinelemeli olarak ikiye böler (ve ikiye bölünmesi çok küçük bir alana neden olur). Bu noktada istenen blok kullanıcıya iade edilir. Burada, bir 7KB bloğu arayışında bölünen 64KB boş alan örneği verilmiştir (Şekil 17.8, sayfa 15).

Bu örnekte, en soldaki 8KB'lik blok tahsis edilmiş (grinin daha koyu tonuyla gösterildiği gibi) ve kullanıcıya geri dönmüştür. Yalnızca 2 boyutlu blogların gücünü vermenize izin verildiğinden, bu şemanın **dahili parçalanmadan (internal fragmentation)** zarar görebileceğini unutmayın.

Arkadaş tahsisinin güzelliği, o blok serbest bırakıldığında Onlarda bulunmasıdır. 8KB'lik bloğu boş listeye geri döndürürken, ayırıcı "arkadaş" 8KB'nin boş olup olmadığını kontrol eder, öyleyse iki bloğu 16KB'lik bir blokta birleştirir. Ayırıcı daha sonra 16KB'lik bloğun arkadaşımın hâlâ boş olup olmadığını kontrol eder, eğer öyleyse bu iki bloğu birleştirir. Bu yinelemeli birleştirme işlemi, ya tüm boş alanı geri yükleyerek ya da bir arkadaşın kullanımda olduğu tespit edildiğinde durarak ağaçta devam eder.



Şekil 17.8: **Bir Yığında Arkadaş Yönetimi Örneği (Example Buddy-managed Heap)**

Arkadaş tahsisinin bu kadar iyi çalışmasının nedeni, belirli bir bloğun arkadaşını belirlemenin basit olmasıdır. Nasıl diye mi soruyorsun? Yukarıda boş alandaki blokların adreslerini düşünün. Yeterince dikkatli düşünürseniz, her bir arkadaş çiftinin adresinin yalnızca bir bit ile farklılık gösterdiğini görürsünüz. Bu bit, arkadaş ağacındaki seviyeye göre belirlenir. Ve böylece ikili arkadaş tahsis şemalarının nasıl çalıştığına dair temel bir fikriniz var. Daha fazla detay için Her zaman olduğu gibi Wilson anketine [W+95] bakın.

Diğer Fikirler (Other Ideas)

Yukarıda açıklanan yaklaşımların çoğunda var olan önemli bir sorun, **ölçeklendirme (scaling)** eksikliğidir. Özellikle, arama listeleri oldukça yavaş olabilir. Bu nedenle gelişmiş ayırıcılar, bu maliyetleri ele almak için daha karmaşık veri yapıları kullanır ve basitliği performansla değiştirirler. Örnekler arasında dengeli ikili ağaçlar, yayvan ağaçlar veya kısmen sıralı ağaçlar bulunur [W+95].

Moda sistemlerin genellikle birden fazla işlemciye sahip olduğu ve çok çekirdekli iş yüklerini çalıştırdığı göz önüne alındığında (kitabın eş zamanlılık bölümünde ayrıntılı olarak öğreneceksiniz), ayırıcıların çok işlemcili sistemlerde iyi çalışmasını sağlamak Fazla çaba harcanması şaşırtıcı değildir. Berger Ve diğerlerinde 2 harika örnek bulunur. [B+00] ve Evans [E06]; ayrıntılar için bunları kontrol edin.

Bunlar, insanların bellek ayırıcılar hakkında zaman içinde sahip oldukları binlerce fikirden yalnızca ikisidir, daha fazlasını merak ediyorsan kendi başınıza okuyun. Bunu başaramazsınız, gerçek dünyanın nasıl bir şey olduğunu anlamanızı sağlamak için glibc ayırıcısının nasıl çalıştığını [S15] okuyun.

17.5 Özet

Bu bölümde, bellek ayırıcılarınının en temel biçimlerini tartıştık. Bu tür ayırıcılar, yazdığınız her C programına ve ayrıca kendi veri yapıları için belleği yöneten temel işletim sistemine bağlı olarak her yerde mevcuttur. Pek çok sistemde olduğu gibi, böyle bir sistemi

oluştururken yapılacak takaslar ve bir ayırıcıya sunulan tam iş yükü hakkında ne kadar çok şey bilirseniz, bu iş yükü için daha iyi çalışacak şekilde ayarlamalar yapabilirsiniz. Çok çeşitli iş yükleri için iyi çalışan, hızlı, alan verimli ve ölçeklenebilir bir ayırıcı yapmak modern bilgisayar sistemlerinde süregelen bir zorluk olmaya devam etmektedir.

References

[B+00] "Hoard: A Scalable Memory Allocator for Multithreaded Applications" by Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. ASPLOS-IX, November 2000. *Berger and company's excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!*

[B94] "The Slab Allocator: An Object-Caching Kernel Memory Allocator" by Jeff Bonwick. USENIX '94. *A cool paper about how to build an allocator for an operating system kernel, and a great example of how to specialize for particular common object sizes.*

[E06] "A Scalable Concurrent malloc(3) Implementation for FreeBSD" by Jason Evans. April, 2006. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf. *A detailed look at how to build a real modern allocator for use in multiprocessors. The "jemalloc" allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.*

[K65] "A Fast Storage Allocator" by Kenneth C. Knowlton. Communications of the ACM, Volume 8:10, October 1965. *The common reference for buddy allocation. Random strange fact: Knuth gives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn't send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing. Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software*[4]*.*

[S15] "Understanding glibc malloc" by Sploitfun. February, 2015. sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/. *A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, September 1995. *An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!*

[4]Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

Ödev (Homework) (Simülasyon) (Simulation)

Malloc.py adlı program, bölümde açıklandığı gibi basit bir boş alan ayırıcısının davranışını öğrenmemizi sağlar. Ayrıntılar için BENİOKU'ya bakın.

Sorular (Questions)

1. Birkaç rastgele ayırma ve serbest bırakma oluşturmak için önce `-n 10 -H 0 -p BEST -s 0` bayraklarıyla çalıştırın. alloc()/free() öğesinin ne döndüreceğini tahmin edebilir misiniz? Herbir istekten sonra boş listenin durumunu tahmin edebilirmisiniz? Zaman içerisinde boş liste hakkında ne fark ettiniz?

   Bellek parçalara bölünmüştür. Sonunda herhangi bir birleşme olmadığından ötürü 1 büyüklüğünde boş alan elde etmiş oluruz.

   ./malloc.py -n 10 -H 0 -p BEST -s 0 -c

   Kodu çalıştırıldığında aşağıdaki çıktıyı elde ederiz.

```
cengizhan@Cengizhan:~$ python3 ./malloc.py -n 10 -H 0 -p BEST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 (searched 4 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

2. Boş listede arama yapmak için (-p WORST) bir kötü uyum ilkesi kullanıldığında sonuçlardaki fark nasıldır? Değişen nedir?

-p WORST ilkesi kullanıldığında alan esikisi gibi ilk başta tıkanmayacağı için sonuç aynı kalacaktır

./malloc.py -n 10 -H 0 -p WORST -s 0 -c

Kodu çalıştırıldığında aşağıdaki çıktıyı elde ederiz.

```
cengizhan@Cengizhan: $ python3 ./malloc.py -n 10 -H 0 -p WORST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy WORST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ]

ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1026 sz:74 ]

ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1033 sz:67 ]
```

3. İlk uyum ilkesi (`-p FIRST`) kullanılırken ne olur? İlk uyum kullanıldığında ne hızlanır?

-p FIRST ilkesi kullanıldığında boş alanların kapsamlı şekilde aranması gerekmediğinden arama daha hızlı hale gelir.
./malloc.py -n 10 -H 0 -p FIRST -s 0 -c
Kodu çalıştırıldığında aşağıdaki çıktıyı elde ederiz.

```
cengizhan@Cengizhan:~$ python3 ./malloc.py -n 10 -H 0 -p FIRST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

4. Yukarıdaki sorular için listenin nasıl sıralı tutulduğu, bazı ilkeler için boş bir konum bulmak amacıyla gereken süreyi etkileyebilir. İlkelerin ve isteklerin nasıl etkili olduğunu görmek için farklı boş liste isteklerini kullanın (`-l ADDRSORT`, `-l SIZESORT+`,`-l SIZESORT-`).

   İlk uyum stratejisi üzerinde hiçbir etkisi yoktur. En iyi uyum ile En kötü uyum sizesort+ ve sizesort- yararlanır. Addrsortun devreye girmesi için birleşme olması gerekir.

   `./malloc.py -p BEST -l SIZESORT+ -c`

   Kodu çalıştırıldığında aşağıdaki çıktıyı elde ederiz.

```
cenglzhan@Cenglzhan:~$ python3 ./malloc.py -p BEST -l SIZESORT+ -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder SIZESORT+
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 (searched 4 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

```
./malloc.py -p WORST -l SIZESORT- -c
```

Kodu çalıştırıldığında aşağıdaki çıktıyı elde ederiz.

```
cenglzhan@Cenglzhan:~$ python3 ./malloc.py -p WORST -l SIZESORT- -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy WORST
listOrder SIZESORT-
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1003 sz:97 ][ addr:1000 sz:3 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1008 sz:92 ][ addr:1000 sz:3 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1008 sz:92 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[3] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1026 sz:74 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1033 sz:67 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
```

5. Boş bir listenin birleştirilmesi oldukça önemli olabilir. Rastgele ayırıcıların sayısını artırın (`-n 1000'e kadar sayın`). Zaman içinde daha büyük ayırma taleplerine ne olur? Birleştirerek ve birleştirmeden çalıştırın (yani, -C bayrağıyla ve bu bayrak olmadan). Sonuçta ne gibi farklılıklar görüyorsunuz? Her durumda boş liste zaman içinde ne kadar büyüktür? Bu durumda listenin sıramalası önemli midir?

   Birleştirme olmadan daha büyük ayırma istekleri NULL değer döndürürve boş listenin boyutu daha büyük olur.

   Listenin sıralaması konusunda ise Adrese göre sıralamak daha iyidir.

   ./malloc.py -n 1000 -r 30 -c -C

   C bayrağı kullanılarak elde edilen çıktının başlangıç kısmı ve son kısmı.

```
cengizhan@cengizhan: $ python3 ./malloc.py -n 1000 -r 30 -c -C
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce True
numOps 1000
range 30
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(8) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1008 sz:92 ]

Free(ptr[0])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]

ptr[1] = Alloc(15) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1015 sz:85 ]

Free(ptr[1])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]

ptr[2] = Alloc(23) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1023 sz:77 ]

Free(ptr[2])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]

ptr[3] = Alloc(22) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1022 sz:78 ]

Free(ptr[3])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]

ptr[4] = Alloc(4) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1004 sz:96 ]

ptr[5] = Alloc(19) returned 1004 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1023 sz:77 ]

Free(ptr[5])
returned 0
Free List [ Size 1 ]: [ addr:1004 sz:96 ]

ptr[6] = Alloc(26) returned 1004 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1030 sz:70 ]
```

```
Free(ptr[534])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:69 ][ addr:1094 sz:6 ]

ptr[535] = Alloc(25) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1025 sz:44 ][ addr:1094 sz:6 ]

ptr[536] = Alloc(12) returned 1025 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1037 sz:32 ][ addr:1094 sz:6 ]

Free(ptr[536])
returned 0
Free List [ Size 2 ]: [ addr:1025 sz:44 ][ addr:1094 sz:6 ]

ptr[537] = Alloc(13) returned 1025 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1038 sz:31 ][ addr:1094 sz:6 ]

Free(ptr[532])
returned 0
Free List [ Size 2 ]: [ addr:1038 sz:55 ][ addr:1094 sz:6 ]

Free(ptr[535])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:25 ][ addr:1038 sz:55 ][ addr:1094 sz:6 ]

Free(ptr[537])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:93 ][ addr:1094 sz:6 ]

ptr[538] = Alloc(27) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1027 sz:66 ][ addr:1094 sz:6 ]

Free(ptr[538])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:93 ][ addr:1094 sz:6 ]

ptr[539] = Alloc(26) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1026 sz:67 ][ addr:1094 sz:6 ]

Free(ptr[533])
returned 0
Free List [ Size 1 ]: [ addr:1026 sz:74 ]

Free(ptr[539])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]

ptr[540] = Alloc(29) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1029 sz:71 ]

Free(ptr[540])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:100 ]
```

./malloc.py -n 1000 -r 30 -c

C bayrağı kullanılmadan elde edilen çıktının başlangıç kısmı ve son kısmı aşağıdaki gibidir.

```
cengizhan@Cengizhan:~$ python3 ./malloc.py -n 1000 -r 30 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 1000
range 30
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(8) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1008 sz:92 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:8 ][ addr:1008 sz:92 ]

ptr[1] = Alloc(15) returned 1008 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:8 ][ addr:1023 sz:77 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:8 ][ addr:1008 sz:15 ][ addr:1023 sz:77 ]

ptr[2] = Alloc(23) returned 1023 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:8 ][ addr:1008 sz:15 ][ addr:1046 sz:54 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:8 ][ addr:1008 sz:15 ][ addr:1023 sz:23 ][ addr:1046 sz:54 ]

ptr[3] = Alloc(22) returned 1023 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:8 ][ addr:1008 sz:15 ][ addr:1045 sz:1 ][ addr:1046 sz:54 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:8 ][ addr:1008 sz:15 ][ addr:1023 sz:22 ][ addr:1045 sz:1 ][ addr:1046 sz:54 ]

ptr[4] = Alloc(4) returned 1000 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1004 sz:4 ][ addr:1008 sz:15 ][ addr:1023 sz:22 ][ addr:1045 sz:1 ][ addr:1046 sz:54 ]

ptr[5] = Alloc(19) returned 1023 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1004 sz:4 ][ addr:1008 sz:15 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:54 ]

Free(ptr[5])
returned 0
Free List [ Size 6 ]: [ addr:1004 sz:4 ][ addr:1008 sz:15 ][ addr:1023 sz:19 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:54 ]

ptr[6] = Alloc(26) returned 1046 (searched 6 elements)
Free List [ Size 6 ]: [ addr:1004 sz:4 ][ addr:1008 sz:15 ][ addr:1023 sz:19 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1072 sz:28 ]
```

```
Free(ptr[729])
returned 0
Free List [ Size 31 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ][ addr:1040 sz:1 ]
[ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ addr:1064 sz:7 ][ ad
dr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

Free(ptr[728])
returned 0
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[730] = Alloc(5) returned 1018 (searched 32 elements)
Free List [ Size 31 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ][ addr:1040 sz:1 ]
[ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ addr:1064 sz:7 ][ ad
dr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

Free(ptr[730])
returned 0
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[731] = Alloc(29) returned -1 (searched 32 elements)
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[732] = Alloc(28) returned -1 (searched 32 elements)
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[733] = Alloc(15) returned -1 (searched 32 elements)
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[734] = Alloc(12) returned -1 (searched 32 elements)
Free List [ Size 32 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1008 sz:8 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ]
[ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ ad
dr:1064 sz:7 ][ addr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[735] = Alloc(8) returned 1008 (searched 32 elements)
Free List [ Size 31 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ][ addr:1040 sz:1 ]
[ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ addr:1064 sz:7 ][ ad
dr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]

ptr[736] = Alloc(26) returned -1 (searched 31 elements)
Free List [ Size 31 ]: [ addr:1000 sz:4 ][ addr:1004 sz:4 ][ addr:1016 sz:2 ][ addr:1018 sz:5 ][ addr:1023 sz:9 ][ addr:1032 sz:1 ][ addr:1033 sz:1 ][ addr:1034 sz:1 ][ addr:1035 sz:5 ][ addr:1040 sz:1 ]
[ addr:1041 sz:1 ][ addr:1042 sz:3 ][ addr:1045 sz:1 ][ addr:1046 sz:10 ][ addr:1056 sz:1 ][ addr:1057 sz:1 ][ addr:1058 sz:1 ][ addr:1059 sz:2 ][ addr:1061 sz:1 ][ addr:1062 sz:2 ][ addr:1064 sz:7 ][ ad
dr:1071 sz:1 ][ addr:1072 sz:10 ][ addr:1082 sz:1 ][ addr:1083 sz:1 ][ addr:1084 sz:1 ][ addr:1085 sz:1 ][ addr:1086 sz:4 ][ addr:1090 sz:6 ][ addr:1096 sz:1 ][ addr:1097 sz:3 ]
```

6. Ayrılan kesir yüzdesini -P 50'den yüksek olarak değiştirdiğinizde ne olur? 100'e yaklaştıkça ayırıcılara ne olur? Yüzde 0'a yaklaştığında ne olur?
   50'den yüksekse sistem sonunda dolar.0'a yaklaştıkça ise kesir yüzdesi hala %50 olarak kabul edilir. Ayrılacak başka alan olmadığından tüm işaretçiler serbest bırakılır.
   100'e yakalaştıkça elde ettiğimiz çıktının başı ve sonu.

```
cenglzhan@cenglzhan:~$ python3 ./malloc.py -c -n 1000 -P 100
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 1000
range 10
percentAlloc 100
allocList
compute True

ptr[0] = Alloc(8) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1008 sz:92 ]

ptr[1] = Alloc(3) returned 1008 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1011 sz:89 ]

ptr[2] = Alloc(5) returned 1011 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1016 sz:84 ]

ptr[3] = Alloc(4) returned 1016 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1020 sz:80 ]

ptr[4] = Alloc(6) returned 1020 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1026 sz:74 ]

ptr[5] = Alloc(6) returned 1026 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1032 sz:68 ]

ptr[6] = Alloc(8) returned 1032 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1040 sz:60 ]

ptr[7] = Alloc(3) returned 1040 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1043 sz:57 ]

ptr[8] = Alloc(10) returned 1043 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1053 sz:47 ]

ptr[9] = Alloc(10) returned 1053 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1063 sz:37 ]

ptr[10] = Alloc(8) returned 1063 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1071 sz:29 ]

ptr[11] = Alloc(7) returned 1071 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1078 sz:22 ]

ptr[12] = Alloc(2) returned 1078 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1080 sz:20 ]
```

```
ptr[982] = Alloc(8) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[983] = Alloc(5) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[984] = Alloc(3) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[985] = Alloc(3) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[986] = Alloc(1) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[987] = Alloc(10) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[988] = Alloc(2) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[989] = Alloc(4) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[990] = Alloc(1) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[991] = Alloc(1) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[992] = Alloc(10) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[993] = Alloc(4) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[994] = Alloc(4) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[995] = Alloc(4) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[996] = Alloc(6) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[997] = Alloc(7) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[998] = Alloc(1) returned -1 (searched 0 elements)
Free List [ Size 0 ]:

ptr[999] = Alloc(4) returned -1 (searched 0 elements)
Free List [ Size 0 ]:
```

0'a yaklaştığımızda elde ettiğimiz kodun başı ve sonu.

```
cengizhan@cengizhan: $ python3 ./malloc.py -C -n 1000 -P 1
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 1000
range 10
percentAlloc 1
allocList
compute True

ptr[0] = Alloc(5) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1005 sz:95 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:5 ][ addr:1005 sz:95 ]

ptr[1] = Alloc(2) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1002 sz:3 ][ addr:1005 sz:95 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:95 ]

ptr[2] = Alloc(9) returned 1005 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1014 sz:86 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]

ptr[3] = Alloc(2) returned 1000 (searched 4 elements)
Free List [ Size 3 ]: [ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:9 ][ addr:1014 sz:86 ]

ptr[4] = Alloc(4) returned 1005 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1009 sz:5 ][ addr:1014 sz:86 ]

Free(ptr[4])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:2 ][ addr:1002 sz:3 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:86 ]

ptr[5] = Alloc(2) returned 1000 (searched 5 elements)
Free List [ Size 4 ]: [ addr:1002 sz:3 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:86 ]
```

```
Free(ptr[494])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]

ptr[495] = Alloc(10) returned 1063 (searched 22 elements)
Free List [ Size 21 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1073 sz:27 ]

Free(ptr[495])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]

ptr[496] = Alloc(10) returned 1063 (searched 22 elements)
Free List [ Size 21 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1073 sz:27 ]

Free(ptr[496])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]

ptr[497] = Alloc(1) returned 1000 (searched 22 elements)
Free List [ Size 21 ]: [ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ][ addr:1031 sz:2 ]
[ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ addr:1073 sz:27 ]

Free(ptr[497])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]

ptr[498] = Alloc(2) returned 1002 (searched 22 elements)
Free List [ Size 21 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ][ addr:1031 sz:2 ]
[ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ addr:1073 sz:27 ]

Free(ptr[498])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]

ptr[499] = Alloc(5) returned 1009 (searched 22 elements)
Free List [ Size 21 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ][ addr:1031 sz:2 ]
[ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ addr:1073 sz:27 ]

Free(ptr[499])
returned 0
Free List [ Size 22 ]: [ addr:1000 sz:1 ][ addr:1001 sz:1 ][ addr:1002 sz:2 ][ addr:1004 sz:1 ][ addr:1005 sz:4 ][ addr:1009 sz:5 ][ addr:1014 sz:6 ][ addr:1020 sz:1 ][ addr:1021 sz:3 ][ addr:1024 sz:7 ]
[ addr:1031 sz:2 ][ addr:1033 sz:7 ][ addr:1040 sz:1 ][ addr:1041 sz:1 ][ addr:1042 sz:1 ][ addr:1043 sz:8 ][ addr:1051 sz:1 ][ addr:1052 sz:1 ][ addr:1053 sz:9 ][ addr:1062 sz:1 ][ addr:1063 sz:10 ][ ad
dr:1073 sz:27 ]
```

7. Yüksek oranda parçalanmış bir boş alan oluşturmak için ne tür spesifik isteklerde bulunabilirsiniz? Parçalanmış boş listeler oluşturmak için -A bayrağını kullanın ve farklı ilke ve seçeneklerin boş listenin organizasyonunu nasıl değiştirdiğini görün.
Adres sıralama ve birleştirme açıkken bu işlem mümkün değildir.
./malloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4
Kodunu çalıştırdığımızda elde ettiğimiz çıktı aşağıdaki gibidir.

```
cengizhan@Cengizhan: $ python3 ./malloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList +10,+15,+20,+25,+30,-0,-1,-2,-3,-4
compute True

ptr[0] = Alloc(10) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1010 sz:90 ]

ptr[1] = Alloc(15) returned 1010 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1025 sz:75 ]

ptr[2] = Alloc(20) returned 1025 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1045 sz:55 ]

ptr[3] = Alloc(25) returned 1045 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1070 sz:30 ]

ptr[4] = Alloc(30) returned 1070 (searched 1 elements)
Free List [ Size 0 ]:

Free(ptr[0])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:10 ]

Free(ptr[1])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:10 ][ addr:1010 sz:15 ]

Free(ptr[2])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:10 ][ addr:1010 sz:15 ][ addr:1025 sz:20 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:10 ][ addr:1010 sz:15 ][ addr:1025 sz:20 ][ addr:1045 sz:25 ]

Free(ptr[4])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:10 ][ addr:1010 sz:15 ][ addr:1025 sz:20 ][ addr:1045 sz:25 ][ addr:1070 sz:30 ]
```

./mallloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4 -l SIZESORT+ -C
Kodunu çalıştırdığımızda elde ettiğimiz çıktı aşağıdaki gibidir.

```
cengizhan@cengizhan: $ python3 ./malloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4 -l SIZESORT+ -C
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder SIZESORT+
coalesce True
numOps 10
range 10
percentAlloc 50
allocList +10,+15,+20,+25,+30,-0,-1,-2,-3,-4
compute True

ptr[0] = Alloc(10) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1010 sz:90 ]

ptr[1] = Alloc(15) returned 1010 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1025 sz:75 ]

ptr[2] = Alloc(20) returned 1025 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1045 sz:55 ]

ptr[3] = Alloc(25) returned 1045 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1070 sz:30 ]

ptr[4] = Alloc(30) returned 1070 (searched 1 elements)
Free List [ Size 0 ]:

Free(ptr[0])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:10 ]

Free(ptr[1])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:25 ]

Free(ptr[2])
returned 0
Free List [ Size 2 ]: [ addr:1025 sz:20 ][ addr:1000 sz:25 ]

Free(ptr[3])
returned 0
Free List [ Size 3 ]: [ addr:1025 sz:20 ][ addr:1000 sz:25 ][ addr:1045 sz:25 ]

Free(ptr[4])
returned 0
Free List [ Size 3 ]: [ addr:1025 sz:20 ][ addr:1000 sz:25 ][ addr:1045 sz:55 ]
```

./malloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4 -l SIZESORT- -C
Kodunu çalıştırdığımızda elde ettiğimiz çıktı aşağıdaki gibidir.

```
cengizhan@cengizhan: $ python3 ./malloc.py -c -A +10,+15,+20,+25,+30,-0,-1,-2,-3,-4 -l SIZESORT- -C
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder SIZESORT-
coalesce True
numOps 10
range 10
percentAlloc 50
allocList +10,+15,+20,+25,+30,-0,-1,-2,-3,-4
compute True

ptr[0] = Alloc(10) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1010 sz:90 ]

ptr[1] = Alloc(15) returned 1010 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1025 sz:75 ]

ptr[2] = Alloc(20) returned 1025 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1045 sz:55 ]

ptr[3] = Alloc(25) returned 1045 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1070 sz:30 ]

ptr[4] = Alloc(30) returned 1070 (searched 1 elements)
Free List [ Size 0 ]:

Free(ptr[0])
returned 0
Free List [ Size 1 ]: [ addr:1000 sz:10 ]

Free(ptr[1])
returned 0
Free List [ Size 2 ]: [ addr:1010 sz:15 ][ addr:1000 sz:10 ]

Free(ptr[2])
returned 0
Free List [ Size 3 ]: [ addr:1025 sz:20 ][ addr:1010 sz:15 ][ addr:1000 sz:10 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1045 sz:25 ][ addr:1025 sz:20 ][ addr:1010 sz:15 ][ addr:1000 sz:10 ]

Free(ptr[4])
returned 0
Free List [ Size 5 ]: [ addr:1070 sz:30 ][ addr:1045 sz:25 ][ addr:1025 sz:20 ][ addr:1010 sz:15 ][ addr:1000 sz:10 ]
```